

动态规划

斐波那契数列系列问题

思路：如果只是求第 n 项，以下三种方法，从坏到好：

解法一：完全抄定义

```
1 | def f(n):
2 |     if n==1 or n==2:
3 |         return 1
4 |     return f(n-1)+f(n-2)
```

该解法效率极低，因为每次求一个 $f(n)$ 都要求出 $f(n-1 \dots 1)$ ，出现了大量的重复计算

解法2：

```
1 | def f1(n):
2 |     if n==1 or n==2:
3 |         return 1
4 |     l=[0]*n          #保存结果
5 |     l[0],l[1]=1,1    #赋初值
6 |     for i in range(2,n):
7 |         l[i]=l[i-1]+l[i-2]  #直接利用之前结果
8 |     return l[-1]
```

可以看出，时间 $O(n)$ ，空间 $O(n)$ 。

该解法保存了中间值，避免了重复计算，但如果只需要求出第 n 项，那么是不是空间有点浪费呢？所以有了下面的解法：

解法3：

```
1 | def f2(n):
2 |     a,b=1,1
3 |     for i in range(n-1): 因为前两项已经给出，所
4 |         a,b=b,a+b      以只要计算n-2次就好
5 |     return a
```

a 往后移动一位，后一位正好是 b；

b 也往后移动一位，后一位是 a + b 的和

青蛙跳台阶和爬楼梯问题：（只能跳、爬一阶或两阶）

动态规划思想：到第 n 阶的情况只有两种情况：

从第 n-1 阶爬上来，求爬到 n-1 阶的方法种数；

从第 $n-2$ 阶爬上来，求爬到 $n-2$ 阶的方法种数；

- 1、子问题规划： $dp[n]$ 爬到第 n 阶台阶所需的方法数
- 2、Base case: $dp[0] = 1, dp[1] = 1, dp[2] = 2$
- 3、状态转移方程： $dp[n] = dp[n - 1] + dp[n - 2]$

变态跳台阶

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级……它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法

思路：与普通的跳台阶问题相比，这里的跳数不再是 1 和 2，那么青蛙跳上最后一阶台阶的起点可以是 $0 \sim n-1$ ，那么状态转移方程为 $f[n] = f[0] + f[1] + \dots + f[n - 1]$ 。

- 1、子问题规划： $dp[n]$ 跳上第 n 阶台阶的方法数
- 2、Base case : $dp[0] = 1, dp[1] = 1, dp[2] = 2$
- 3、状态转移方程： $dp[n] = \text{sum}(dp[0]+dp[1]+\dots+dp[n-1])$

```
class Solution:  
    def jumpFloorII(self, number):  
        if number == 1:  
            return 1  
        if number == 2:  
            return 2  
        else:  
            dp = [1, 1, 2] # dp[n] 表示跳上第 n 阶台阶的方法数  
            for i in range(3, number + 1):  
                dp.append(sum(dp)) # 可能从前面上任何一个台阶上一步跳上来，所以将所有的情况加起来即可  
            return dp[-1]
```

我们可以用 2×1 的小矩形横着或者竖着去覆盖更大的矩形。

请问用 n 个 2×1 的小矩形无重叠地覆盖一个 $2 \times n$ 的大矩形，总共有多少种方法？

动态规划思想：覆盖到长度为 n 的地方有两种情况：

在第 n 个位置竖着放一个小矩形，求前面 $n-1$ 个位置放矩形的方法数；

从第 $n-1$ 个位置横着放两个小矩形，求前面 $n-2$ 个位置放矩形的方法数；

- 1、子问题规划： $dp[n]$ 覆盖 $2 \times n$ 的大矩形的方法
- 2、Base case : $dp[0] = 0, dp[1] = 1, dp[2] = 1$
- 3、状态转移方程为： $dp[n] = dp[n - 2] + dp[n - 1]$

解码方法

91. 解码方法

难度 中等 470 收藏 分享 切换为英文 关注 反馈

一条包含字母 A-Z 的消息通过以下方式进行了编码：

```
'A' -> 1  
'B' -> 2  
...  
'Z' -> 26
```

给定一个只包含数字的非空字符串，请计算解码方法的总数。

示例 1：

```
输入: "12"  
输出: 2  
解释: 它可以解码为 "AB" (1 2) 或者 "L" (12)。
```

示例 2：

```
输入: "226"  
输出: 3  
解释: 它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6)。
```

动态规划思想：第 n 个数字字符转换完成字母分为两种情况：

第 n 个位置，自身转化为字母

第 n-1 个位置加第 n 个位置，转换为字母（两位数不能大于 26）

1、子问题规划：dp[n] 表示长度为 n 的数字串能够转化的字母串种数

2、Base case：dp[i] = 1

3、状态转移方程：dp[i] = dp[i-1] + dp[i - 2] if int(s[i-1:i+1]) < 26 else dp[i-1]

```
class Solution:  
    def numDecodings(self, s: str) -> int:  
        N = len(s)  
        dp = [1] * N  
        for i in range(1, N):  
            if s[i] == '0' and (s[i - 1] not in ['1', '2']): # 排除两种走法都不能到的情况  
                return 0  
            elif s[i] == '0': # 当前位为0, 那么只能走两步过来  
                dp[i] = dp[i - 2]  
            elif s[i - 1] == '1' or s[i - 1] == '2' and s[i] <= '6': # 符合两种情况的条件  
                dp[i] = dp[i - 1] + dp[i - 2]  
            else: # 剩下的只能走一步过来  
                dp[i] = dp[i - 1]  
        return dp[-1] if s[0] != '0' else 0 # 为0的话直接就不能动弹了
```

剪绳子

题目描述

给你一根长度为n的绳子，请把绳子剪成整数长的m段（m、n都是整数， $n > 1$ 并且 $m > 1$ ， $m \leq n$ ），每段绳子的长度记为 $k[1], \dots, k[m]$ 。请问 $k[1] \times \dots \times k[m]$ 可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

输入描述:

输入一个数n，意义见题面。（ $2 \leq n \leq 60$ ）

输出描述:

输出答案。

示例1

输入	输出
8	18

思路：这道题有一个隐含的数学原理，那就是当绳子大于3的时候，将绳子都拆分成1\2\3长度段时，长度段长度乘积是最大的。

1、子问题规划： $dp[n]$ 表示长度为n的绳子的最大乘积

2、Base case: $dp[1] = 1, dp[2] = 2, dp[3] = 3$

3、状态转移方程： $dp[n] = \max (dp[n - 3] * 3 + dp[n - 2] * 2 + dp[n - 1] * 1)$

```
class Solution:
    def cuttingRope(self, n: int) -> int:
        if n == 2:
            return 1
        if n == 3: # 因为必须要剪
            return 2
        dp = [0] * (n + 1) # 长度为L的绳子的剪完后的最大乘积
        dp[1], dp[2], dp[3] = 1, 2, 3
        for i in range(4, n + 1):
            dp[i] = max(dp[i - 1] * 1, dp[i - 2] * 2, dp[i - 3] * 3)
        return dp[n]
```

注意：这里求的就是方法数，所有不能把所有的情况加起来，而是选择其中最优的一种情况。

组合总和 IV

377. 组合总和 IV

难度 中等 184 收藏 分享 切换为英文 关注 反馈

给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。

示例：

```
nums = [1, 2, 3]
target = 4
```

所有可能的组合为：

```
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
```

请注意，顺序不同的序列被视作不同的组合。

因此输出为 7。

进阶：

如果给定的数组中含有负数会怎么样？

问题会产生什么变化？

我们需要在题目中添加什么限制来允许负数的出现？

思路：斐波那契问题和背包问题很类似，而这个明显重复的有很多组合中有都是组合中元素是相同的，只是顺序不同而已，故还是属于斐波那契问题。

```
class Solution:
    def combinationSum4(self, nums: [int], target: int) -> int:
        dp = [0] * (target + 1)
        for x in nums:
            if x <= target: # 步长不能超过阶梯总数
                dp[x] = 1 # base case就相当于斐波那契数列的前面两个1
        for i in range(target + 1):
            for j in nums: # 遍历能到达该位置的点
                if i - j >= 0:
                    dp[i] += dp[i - j]
        return dp[target]
```

背包问题

基础概念

0/1 背包问题：

概念：每个物品只能取用一次，物品重量 $weight[i]$ ，物品价值 $value[j]$

$dp[i][j]$ 表示容量为 j 的背包装前 i 个物品的最大价值。

二维状态转移方程：

```
dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]] + value[i])
```

```

if __name__ == '__main__':
    weight = [2, 3, 4, 5]
    value = [3, 4, 5, 6]
    N = len(weight)
    capacity = 8
    dp = [[0] * (capacity + 1) for _ in range(N + 1)] # dp[i][j]表示容量为j的背包放置前i个物品的最大价值和
    for i in range(1, N + 1): # 背包容量和物品数目都不能为0
        for j in range(1, capacity + 1): # 这里顺序无所谓，因为需要的数据存在上一行
            if j >= weight[i - 1]:
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i - 1]] + value[i - 1])
            else:
                dp[i][j] = dp[i - 1][j] # 放不下第i个物品，就只能放到第i - 1个物品
    print(dp[-1][-1])

```

一维状态转移方程：

$$dp[j] = \max(dp[j-1], dp[j - weight[i]] + value[i])$$

$dp[j]$ 表示的是当背包容量为 j 时，装前 i 个物品的最大价值

注意：使用一维数组求解，为保证每个物体只取一次，应从后往前赋值。

```

if __name__ == '__main__':
    # 一维解法
    weight = [2, 3, 4, 5]
    value = [3, 4, 5, 6]
    N = len(weight)
    capacity = 8
    dp = [0] * (capacity + 1) # dp[j]表示为容量为j的背包所能放置的物品最大价值和
    for i in range(N):
        for j in range(capacity, weight[i] - 1, -1): # 只取容量大于当前物品的背包
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i])
    print(dp[capacity])

```

完全背包问题：（每个物体可以去无限个）

思路：其实只需将第二个循环的顺序调转过来即可，因为下一个位置的得出用到了前面的位置的值。

```

if __name__ == '__main__':
    # 一维解法
    weight = [2, 3, 4, 5]
    value = [3, 4, 5, 6]
    N = len(weight)
    capacity = 8
    dp = [0] * (capacity + 1) # dp[j]表示为容量为j的背包所能放置的物品最大价值和
    for i in range(N):
        for j in range(weight[i], capacity + 1): # 只取容量大于当前物品的背包
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i])
    print(dp[capacity])

```

多重背包问题：（每个物体只能取有限个）

思路：将多个物体拆成一个一个物体，用 01 背包求解

题目

购物单（多重背包）

题目描述

王强今天很开心，公司发给N元的年终奖。王强决定把年终奖用于购物，他把想买的物品分为两类：主件与附件，附件是从属于某个主件的，下表就是一些主件与附件的例子：

主件	附件
电脑	打印机，扫描仪
书柜	图书
书桌	台灯，文具
工作椅	无

如果要买归类为附件的物品，必须先买该附件所属的主件。每个主件可以有 0 个、 1 个或 2 个附件。附件不再有从属于自己的附件。王强想买的东西很多，为了不超出预算，他把每件物品规定了一个重要度，分为 5 等：用整数 1 ~ 5 表示，第 5 等最重要。他还从因特网上查到了每件物品的价格（都是 10 元的整数倍）。他希望在不超过 N 元（可以等于 N 元）的前提下，使每件物品的价格与重要度的乘积的总和最大。

设第 j 件物品的价格为 $v[j]$ ，重要度为 $w[j]$ ，共选中了 k 件物品，编号依次为 j_1, j_2, \dots, j_k ，则所求的总和为：
 $v[j_1]*w[j_1]+v[j_2]*w[j_2]+\dots+v[j_k]*w[j_k]$ 。（其中 * 为乘号）

请你帮助王强设计一个满足要求的购物单。

输入描述:

输入的第 1 行，为两个正整数，用一个空格隔开：N m
(其中 N (<32000) 表示总钱数，m (<60) 为希望购买物品的个数。)

从第 2 行到第 m+1 行，第 j 行给出了编号为 $j-1$ 的物品的基本数据，每行有 3 个非负整数 v p q

(其中 v 表示该物品的价格 ($v < 10000$)，p 表示该物品的重要度 (1 ~ 5)，q 表示该物品是主件还是附件。如果 $q=0$ ，表示该物品为主件，如果 $q>0$ ，表示该物品为附件，q 是所属主件的编号)

输出描述:

输出文件只有一个正整数，为不超过总钱数的物品的价格与重要度乘积的总和的最大值 (<200000)。

示例1

```
输入
1000 5
800 2 0
400 5 1
400 5 1
300 5 1
400 3 0
500 2 0

输出
2200
```

思路：这道题主要还是 0/1 背包问题，但是需要分组，分组的依据是按照

1、主件 2、主件+附件 1 3、主件+附件 2 4、主件+双附件。

思路正确：多重背包的思想，可参照 CSDN 算法笔记

```
total, row = map(int, input().split())
total //= 10    # 同倍数的缩小背包和物品的重量
w = [[0 for _ in range(4)] for _ in range(row)]
v = [[0 for _ in range(4)] for _ in range(row)]
flag = []
for i in range(row):
    x, y, z = map(int, input().split())
    x //= 10
    if z == 0:
        w[i][0] = x
        v[i][0] = x * y
    else:
        if z not in flag:
            w[z-1][1] = w[z - 1][0] + x    # 主件加第一个附件
            v[z-1][1] = v[z - 1][0] + x * y
            flag.append(z)    # 找到了第一个附件
        else:
            w[z-1][2] = w[z - 1][0] + x
            v[z-1][2] = v[z - 1][0] + x * y
            w[z-1][3] = w[z - 1][1] + x
            v[z-1][3] = v[z - 1][1] + x * y
dp = [0 for _ in range(total + 1)]    # 背包容量为j，在前i个物品中选择，能够放下且价值最大的物品价值和。
for i in range(len(w)):
    for j in range(total, -1, -1):
        for k in range(4):
            if w[i][k] <= j:
                dp[j] = max(dp[j], v[i][k] + dp[j - w[i][k]])
print(dp[total] * 10)
```

零钱兑换（完全背包）

322. 零钱兑换

难度 中等 664 收藏 分享 切换为英文 关注 反馈

给定不同面额的硬币 coins 和一个总金额 amount，编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

示例 1:

```
输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1
```

示例 2:

```
输入: coins = [2], amount = 3
输出: -1
```

说明:

你可以认为每种硬币的数量是无限的。

1、子问题规划： $dp[n]$ 表示凑成总金额为 n 所需的最少硬币个数， $coins[i]$ 代表第 i 个硬币面值。

2、base case: $dp[0] = 0$ ， $dp[n]$ 初始化为无限大，因为有些金额是凑不出来的

3、状态转移方程： $dp[n] = \min (dp[n], 1 + dp[n - coins[i]])$

```

class Solution:
    def coinChange(self, coins: [int], amount: int) -> int:
        dp = [sys.maxsize] * (amount + 1) # dp[j]表示总金额为j时使用前i个硬币使用的最少硬币个数
        dp[0], N = 0, len(coins) # 总金额为0时是凑不出来的
        for i in range(N):
            for j in range(coins[i], amount + 1): # 完全背包，顺序遍历
                dp[j] = min(dp[j], dp[j - coins[i]] + 1)
        return dp[-1] if dp[-1] != sys.maxsize else -1

```

零钱兑换 2 (完全背包)

518. 零钱兑换 II

难度 中等 215 收藏 分享 切换为英文 关注 反馈

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

示例 1:

```

输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1

```

示例 2:

```

输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额2的硬币不能凑成总金额3。

```

示例 3:

```

输入: amount = 10, coins = [10]
输出: 1

```

思路:明显看出每个组合唯一,那就是背包问题,不是斐波那契问题了.

```

class Solution:
    def change(self, amount: int, coins: [int]) -> int:
        dp = [0] * (amount + 1) # dp[j]表示背包容量为j时,用前i个硬币能够装满背包的组合数.
        dp[0] = 1 # 背包容量为0时,组合数为1
        for i in range(len(coins)):
            for j in range(coins[i], amount + 1):
                dp[j] += dp[j - coins[i]]
        return dp[-1]

```

dp[n]不是答案的最值问题

一维：

最大和子数组

53. 最大子序和

难度 简单 2096 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 `nums`，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例：

输入： [-2,1,-3,4,-1,2,1,-5,4],
输出： 6
解释： 连续子数组 [4,-1,2,1] 的和最大，为 6。

1、子问题规划：`dp[n]`表示以 `nums[n]`结尾的子数组最大和

2、Base case：`dp[0] = nums[0]`，其余初始化为 `-sys.maxsize`, `maxSize = -sys.maxsize`

3、状态转移方程：

$$\begin{aligned} dp[i] &= \max(dp[i-1] + nums[i], nums[i]) \\ maxSum &= \max(maxSum, dp[i]) \end{aligned}$$

```
import sys
class Solution:
    def maxSubArray(self, nums):
        N, maxSum = len(nums), -sys.maxsize
        # dp = [-sys.maxsize] * N
        # dp[0] = nums[0]
        # maxSum = max(dp[0], maxSum) # 位置0也要比较
        pre = nums[0]
        maxSum = max(pre, maxSum)
        for i in range(1, N):
            # dp[i] = max(dp[i - 1] + nums[i], nums[i])
            # maxSum = max(maxSum, dp[i])
            pre = max(pre + nums[i], nums[i])
            maxSum = max(maxSum, pre)
        return maxSum
```

最小和子数组

给定一个整数数组，本例将找到一个一个具有最小和的子数组，返回最小和。

1、子问题规划：`dp[n]`以 `nums[n]`结尾的最小和子数组

2、Base case：`dp[0] = nums[0]`，其余初始化为 `sys.maxsize`, `minSum = sys.maxsize`

3、状态转移方程：

$$\begin{aligned} dp[i] &= \min(dp[i - 1] + nums[i], nums[i]) \\ minSum &= \min(minSum, dp[i]) \end{aligned}$$

```
class Solution:
    def minSubArray(self, nums):
        N, minSum = len(nums), sys.maxsize
        dp = [sys.maxsize] * N # dp[i]是nums[i]结尾的最小子数组和
        # dp[0] = nums[0]
        preSum = nums[0]
        for i in range(1, N):
            # dp[i] = min(dp[i - 1] + nums[i], nums[i])
            preSum = min(preSum + nums[i], nums[i])
            minSum = min(minSum, preSum)
        return minSum
```

两个不重叠子数组最大和

1031. 两个非重叠子数组的最大和

难度 中等 61 收藏 分享 切换为英文 关注 反馈

给出非负整数数组 A ，返回两个非重叠（连续）子数组中元素的最大和，子数组的长度分别为 L 和 M 。（这里需要澄清的是，长为 L 的子数组可以出现在长为 M 的子数组之前或之后。）

从形式上看，返回最大的 V ，而 $V = (A[i] + A[i+1] + \dots + A[i+L-1]) + (A[j] + A[j+1] + \dots + A[j+M-1])$ 并满足下列条件之一：

- $0 \leq i < i + L - 1 < j < j + M - 1 < A.length$, 或
- $0 \leq j < j + M - 1 < i < i + L - 1 < A.length$.

示例 1：

输入： $A = [0, 6, 5, 2, 2, 5, 1, 9, 4]$, $L = 1$, $M = 2$
输出：20
解释：子数组的一种选择中， $[9]$ 长度为 1, $[6, 5]$ 长度为 2。

示例 2：

输入： $A = [3, 8, 1, 3, 2, 1, 8, 9, 0]$, $L = 3$, $M = 2$
输出：29
解释：子数组的一种选择中， $[3, 8, 1]$ 长度为 3, $[8, 9]$ 长度为 2。

示例 3：

输入： $A = [2, 1, 5, 6, 0, 9, 5, 0, 3, 8]$, $L = 4$, $M = 3$
输出：31
解释：子数组的一种选择中， $[5, 6, 0, 9]$ 长度为 4, $[0, 3, 8]$ 长度为 3。

提示：

1. $L \geq 1$
2. $M \geq 1$
3. $L + M \leq A.length \leq 1000$
4. $0 \leq A[i] \leq 1000$

1、子问题规划：sumL[n]到 n 位置的长度为 L 的最大和子数组

totalSum[m]到 m 位置的两子数组的最大和

2、base case:

```

sumL[L - 1] = prefix[L - 1],
dp[L + M - 1] = prefix[L + M - 1]

```

3、状态转移方程：

```

sumL[i] = max(sum[i - 1], prefix[i] - prefix[i - L])
dp[i] = max(dp[i-1], sumL[i - M] + prefix[i] - prefix[i - M])

```

```

class Solution:
    def maxSumTwoNoOverlap(self, A: [int], L: int, M: int) -> int:
        N = len(A)
        preSum = [0] * (N + 1)
        for i in range(1, N + 1):
            preSum[i] = preSum[i - 1] + A[i - 1] # preSum的坐标比原数组大1
        LM = self.sumOfTwoArr(preSum, L, M)
        ML = self.sumOfTwoArr(preSum, M, L)
        return max(LM, ML)

    def sumOfTwoArr(self, preSum, L, M):
        N = len(preSum)
        Lmax, Max = [0] * N, [0] * N # Lmax[i]表示A[:i]长度为L的最大和, Max[i]表示A[:i]长度为L+M的最大和
        for i in range(L, N): # 求长度为L的最大子数组的正确方式
            Lmax[i] = max(Lmax[i - 1], preSum[i] - preSum[i - L])
        for j in range(L + M, N):
            Max[i] = max(Max[i - 1], Lmax[j] + preSum[j] - preSum[j - M])
        return Max[-1]

```

最大乘积子数组

152. 乘积最大子数组

难度 中等 击 626 收藏 分享 切换为英文 关注 反馈

给你一个整数数组 `nums`，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

示例 1:

```

输入: [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。

```

示例 2:

```

输入: [-2,0,-1]
输出: 0
解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。

```

1、子问题规划：positive[i]表示以 `nums[i]` 结尾的乘积最大的连续子数组

`negative[i]` 表示以 `nums[i]` 结尾的乘积最小的连续子数组

2、base case : `positive[0] = nums[0]` , 其余初始化为`-sys.maxsize`

`negative[0] = nums[0]`, 其余初始化为 `sys.maxsize`

```
maxProduct = -sys.maxsize
```

3、状态转移方程：

```
If nums[i] < 0:  
    positive[i-1], negative[i-1] = negative[i-1], positive[i-1]  
    positive[i] = max(positive[i-1]*nums[i], nums[i])  
    negative[i] = max(negative[i-1]*nums[i], nums[i])  
    maxProduct = max(maxProduct, positive[i])
```

4、空间优化：

```
class Solution:  
    def maxProduct(self, nums):  
        N, maxProduct = len(nums), -sys.maxsize  
        if N == 1:  
            return nums[0]  
        # positive = [-sys.maxsize] * N  
        # negative = [sys.maxsize] * N  
        pos, neg = 0, 0  
        for i in range(N):  
            if i == 0:  
                # positive[0], negative[0] = nums[0], nums[0]  
                pos, neg = nums[0], nums[0]  
            else:  
                if nums[i] < 0: # 小于0时乘以最小值反而是最大值，所以要调换顺序  
                    # positive[i-1], negative[i-1] = negative[i-1], positive[i-1]  
                    pos, neg = neg, pos  
                # positive[i] = max(positive[i-1] * nums[i], nums[i])  
                # negative[i] = min(negative[i-1] * nums[i], nums[i])  
                pos = max(pos * nums[i], nums[i])  
                neg = min(neg * nums[i], nums[i])  
                # maxProduct = max(maxProduct, positive[i])  
                maxProduct = max(maxProduct, pos)  
        return maxProduct
```

最长上升子序列

300. 最长上升子序列

难度 中等 778 收藏 分享

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例：

输入： [10,9,2,5,3,7,101,18]

输出： 4

解释： 最长的上升子序列是 [2,3,7,101]，它的长度是 4。

说明：

- 可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
- 你算法的时间复杂度应该为 $O(n^2)$ 。

进阶：你能将算法的时间复杂度降低到 $O(n \log n)$ 吗？

1、子问题规划： $dp[i]$ 表示以 $nums[i]$ 结尾的最长上升子序列的长度

2、Base case：都初始化为 1, $maxLen = 0$

3、状态转移方程：

$$dp[i] = \max(dp[i], dp[j] + 1) \quad 0 \leq j < i \\ maxLen = \max(maxLen, dp[i])$$

```
class Solution:
    def lengthOfLIS(self, nums):
        if nums is None or not nums:
            return 0
        dp = [1] * len(nums) # 以第i个元素结尾的最长上升子序列长度
        MAX, size = 0, len(nums)
        for i in range(size):
            for j in range(i): # 第i之前的所有位置遍历
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)
            MAX = max(MAX, dp[i])
        return MAX
```

最长连续序列

128. 最长连续序列

难度 困难 461 收藏 分享 切换为英文 关注 反馈

给定一个未排序的整数数组，找出最长连续序列的长度。

要求算法的时间复杂度为 $O(n)$ 。

示例：

```
输入: [100, 4, 200, 1, 3, 2]
输出: 4
解释: 最长连续序列是 [1, 2, 3, 4]。它的长度为 4。
```

1、暴力法：将数组存入 hash 表，遍历其中每一个元素，将每个元素自加 1，如果在 hash 表中，如果长度加 1。（这样的话会做了很多重复的加法，例如 1 自加到了 4, 3 自加也到了 4，2 自加也到了 4）。

2、调优：当前元素减一，如果所得值还在 hash 表内，表明迟早会自加到它，那么就跳过即可。

```
class Solution:
    def longestConsecutive(self, nums: [int]) -> int:
        N, hash, maxLen = len(nums), set(), 0
        for i in range(N):
            hash.add(nums[i])
        for i in range(N):
            if nums[i] - 1 in hash:
                continue
            currentLen, currentValue = 1, nums[i] + 1
            while currentValue in hash:
                currentLen += 1
                currentValue += 1
            maxLen = max(maxLen, currentLen)
        return maxLen
```

二维

最大正方形

221. 最大正方形

难度 中等 464 收藏 贡献 提交

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

示例：

输入：

```
1 0 1 0 0  
1 0 1 1 1  
1 1 1 1 1  
1 0 0 1 0
```

输出：4

我的思路：设定一个动态数组 $dp[i][j]$ ，表明以这个点为右下角的最大正方形的边长。

将找最大正方形的问题转化到找最大正方形的边长的问题。如果该点为 1 的话，某点的最大正方形边长等于 $\min(\text{该点上方最大正方形边长}, \text{该点左方最大正方形边长}, \text{该点左上方最大正方形边长}) + 1$ ，如果该点为 0，那么还会是 0，因为无法以该点为右下角找到正方形。

1、子问题规划： $matrix[i][j]$ 表示以 $matrix[i][j]$ 为右下角的最大正方形边长

2、Base case：第一行和第一列自带 base case，第一行和第一列即为其 base case。

3、状态转移方程：

```
if matrix[i][j] == 1:  
    if not (i == 0 or j == 0):  
        matrix[i][j] = min(matrix[i-1][j], matrix[i][j-1], matrix[i-1][j-1])  
    MAX = max(MAX, matrix[i][j])
```

```

class Solution:
    def maximalSquare(self, matrix):
        if len(matrix) == 0 or len(matrix[0]) == 0:
            return 0
        rowNum, colNum, MAX = len(matrix), len(matrix[0]), 0
        for i in range(rowNum):
            for j in range(colNum):
                matrix[i][j] = int(matrix[i][j])
                if matrix[i][j] == 1:
                    if not (i == 0 or j == 0):
                        matrix[i][j] = min(matrix[i - 1][j], matrix[i - 1][j - 1], matrix[i][j - 1]) + 1
                    MAX = max(MAX, matrix[i][j]) # 第一行第一列直接base case
        return MAX * MAX

```

最长重复子数组

718. 最长重复子数组

难度 中等 179 收藏 分享 切换为英文 关注 反馈

给两个整数数组 A 和 B，返回两个数组中公共的、长度最长的子数组的长度。

示例 1：

输入：
A: [1,2,3,2,1]
B: [3,2,1,4,7]
输出：3
解释：
长度最长的公共子数组是 [3, 2, 1]。

说明：

1. 1 <= len(A), len(B) <= 1000
2. 0 <= A[i], B[i] < 100

1、子问题规划：dp[i][j] 表示 A 以 A[i] 结尾和 B 以 B[j] 结尾的最长重复子数组长度

2、Base case：均初始化为 0, maxLen = 0

3、状态转移方程：dp[i][j] = dp[i-1][j-1] + 1 if A[i] == B[j] else 0

$$\text{maxLen} = \max(\text{maxLen}, \text{dp}[i][j])$$

```
class Solution:
    def findLength(self, A: [int], B: [int]) -> int:
        M, N = len(A), len(B)
        maxLen = 0
        dp = [[0] * N for _ in range(M)] # dp[i][j] 表示A以A[i]结尾和B以B[j]结尾的最长湍流子数组长度
        for i in range(M):
            for j in range(N):
                if A[i] == B[j]:
                    if i == 0 or j == 0:
                        dp[i][j] = 1
                    else:
                        dp[i][j] = dp[i - 1][j - 1] + 1
                else:
                    dp[i][j] = 0
                maxLen = max(maxLen, dp[i][j])
        return maxLen
```

多子问题

最长湍流子数组

978. 最长湍流子数组

难度 中等 43 收藏 分享 切换为英文 关注 反馈

当 A 的子数组 $A[i], A[i+1], \dots, A[j]$ 满足下列条件时，我们称其为湍流子数组：

- 若 $i \leq k < j$ ，当 k 为奇数时， $A[k] > A[k+1]$ ，且当 k 为偶数时， $A[k] < A[k+1]$ ；
- 或若 $i \leq k < j$ ，当 k 为偶数时， $A[k] > A[k+1]$ ，且当 k 为奇数时， $A[k] < A[k+1]$ 。

也就是说，如果比较符号在子数组中的每个相邻元素对之间翻转，则该子数组是湍流子数组。

返回 A 的最大湍流子数组的长度。

示例 1：

```
输入：[9, 4, 2, 10, 7, 8, 8, 1, 9]
输出：5
解释：(A[1] > A[2] < A[3] > A[4] < A[5])
```

示例 2：

```
输入：[4, 8, 12, 16]
输出：2
```

示例 3：

```
输入：[100]
输出：1
```

提示：

1. $1 \leq A.length \leq 40000$
2. $0 \leq A[i] \leq 10^9$

1、子问题规划： $up[i]$ 表示以 $A[i]$ 结尾且末尾上升的最长湍流子数组长度

$down[i]$ 表示以 $A[i]$ 结尾且末尾下降的最长湍流子数组长度

2、base case : $up[0] = 1$, $down[0] = 1$, 其余均初始化为 0 , $maxLen = 1$

3、状态转移方程：

```
if A[i-1] < A[i]:
    up[i] = down[i-1] + 1
    down[i] = 1
elif A[i-1] > A[i]:
    down[i] = up[i] + 1
    up[i] = 1
else:
```

```

    up[i] = 1
    down[i] = 1
    maxLen = max(maxLen, up[i], down[i])

```

4、空间优化

```

class Solution:
    def maxTurbulenceSize(self, A: [int]) -> int:
        N, maxLen = len(A), 1
        # up, down = [0] * N, [0] * N
        # up[0], down[0] = 1, 1
        up, down = 1, 1
        for i in range(1, N):
            if A[i - 1] < A[i]:
                # up[i] = down[i-1] + 1
                # down[i] = 1
                up = down + 1
                down = 1
            elif A[i - 1] > A[i]:
                # down[i] = up[i-1] + 1
                # up[i] = 1
                down = up + 1
                up = 1
            else:
                # up[i] = 1
                # down[i] = 1
                up, down = 1, 1
        # maxLen = max(maxLen, up[i], down[i])
        maxLen = max(maxLen, up, down)
        return maxLen

```

dp[n]是答案的最值问题

一维：

切分数组

LCP 14. 切分数组

难度 困难 14 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 `nums`，小李想将 `nums` 切割成若干个非空子数组，使得每个子数组最左边的数和最右边的数的最大公约数大于 1。为了减少他的工作量，请求出最少可以切成多少个子数组。

示例 1：

输入：`nums = [2,3,3,2,3,3]`

输出：2

解释：最优切割为 `[2,3,3,2]` 和 `[3,3]`。第一个子数组头尾数字的最大公约数为 2，第二个子数组头尾数字的最大公约数为 3。

示例 2：

输入：`nums = [2,3,5,7]`

输出：4

解释：只有一种可行的切割：`[2], [3], [5], [7]`

限制：

- `1 <= nums.length <= 10^5`
- `2 <= nums[i] <= 10^6`

1、子问题规划：`dp[n]` 表示当 `nums[n]` 为最右边的数时能切分的最少子数组数

2、Base case：`dp[0] = 1`, 其余均初始化为 `sys.maxsize`

3、状态转移方程：

```
for i in range(N):
    for j in range(i):
        if gcd(nums[j], nums[i]) > 1
            If j == 0:
                dp[[i]] = 1
            Else:
                dp[i] = max(dp[i], dp[j-1] + 1)
        If dp[i] = sys.maxsize:
            dp[i] = dp[i-1] + 1 # 跟任何数的最大公约数都小于等于 1
```

```

import sys

class Solution(object):
    # todo...超时
    def splitArray(self, nums):
        N = len(nums)
        dp = [sys.maxsize] * N # dp[i] 表示以nums[i]结尾能切分的数据能切分的最小子数组
        dp[0] = 1
        for i in range(len(dp)):
            for j in range(i):
                if self.gcd(nums[i], nums[j]) > 1:
                    if j == 0: # 匹配到第一个，直接为1
                        dp[i] = 1
                    else:
                        dp[i] = min(dp[i], dp[j - 1] + 1) # 比较所有可能情况的最小值，核心状态转移方程
                if dp[i] == sys.maxsize: # 没有匹配到任何数
                    dp[i] = dp[i - 1] + 1
        return dp[N - 1]

    def gcd(self, num1, num2):
        while num1 != 0:
            num1, num2 = num2 % num1, num1 # 只需要记住这样就能保证num2能够大于num1，所以要将num2 % num1的结果给num1
        return num2

```

解答错误 22/43

分割回文串 2

132. 分割回文串 II

难度 困难 154 收藏 分享 切换为英文 关注 反馈

给定一个字符串 s ，将 s 分割成一些子串，使每个子串都是回文串。

返回符合要求的最少分割次数。

示例:

输入: "aab"

输出: 1

解释: 进行一次分割就可将 s 分割成 $["aa", "b"]$ 这样两个回文子串。

1、子问题规划 : $dp[n]$ 表示 $s[n]$ 结尾的字符串的最少分割次数

2、Base case : $dp[0] = 0$, $dp = [sys.maxsize] * n$

3、状态转移方程 :

```

if s[:i+1] ishuiwen:
    dp[i] = 0
elif s[j+1:i+1] ishuiwen:

```

$$dp[i] = \min(dp[i], dp[j] + 1)$$

```
class Solution:
    def minCut(self, s: str) -> int:
        N = len(s)
        if N == 0 or N == 1:
            return 0
        dp = [sys.maxsize] * N # dp[i] 表示以 s[i] 结尾的最小分割次数
        dp[0] = 0
        for i in range(N):
            for j in range(i):
                if self.isHuiwen(s[:i+1]): # 奇怪情况
                    dp[i] = 0
                    break
                elif self.isHuiwen(s[j+1:i+1]):
                    dp[i] = min(dp[i], dp[j] + 1)
        return dp[-1]

    def isHuiwen(self, s):
        return s[::-1] == s
```

打家劫舍 1

198. 打家劫舍

难度 简单 896 收藏 分享 切换为英文 关注 反馈

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

示例 1：

```
输入：[1, 2, 3, 1]
输出：4
解释：偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。
偷窃到的最高金额 = 1 + 3 = 4。
```

示例 2：

```
输入：[2, 7, 9, 3, 1]
输出：12
解释：偷窃 1 号房屋（金额 = 2），偷窃 3 号房屋（金额 = 9），接着偷窃 5 号房屋（金额 = 1）。
偷窃到的最高金额 = 2 + 9 + 1 = 12。
```

1、子问题规划： $dp[i]$ 偷前 i 家房屋所能偷窃到的最高金额

2、Base case : $dp[0]=0$, $dp[1]=nums[0]$, 求最高金额 , 所以其余均初始化为 0

3、状态转移方程 : $dp[i] = \max(dp[i-1], dp[i-1] + nums[i])$ # 不偷和偷

4、空间优化 :

```
class Solution:
    def rob(self, nums: [int]) -> int:
        N = len(nums)
        if N == 0:
            return 0
        dp = [0] * (N + 1) # dp[i] 表示打劫前n家商店的最高金额
        dp[1] = nums[0] # 打劫第一家商店
        for i in range(2, N + 1):
            dp[i] = max(dp[i - 1], dp[i - 2] + nums[i - 1])
        return dp[N]
```

其实也只是用到了三个数 , 可以进行空间优化

```
class Solution:
    def rob(self, nums: [int]) -> int:
        N = len(nums)
        # dp = [0] * (N + 1) # dp[i] 表示打劫前n家商店的最高金额
        if N == 0:
            return 0
        if N == 1:
            return nums[0]
        pre, cur = 0, nums[0] # pre表示往前两个数 , cur表示往前一个数
        for i in range(2, N + 1):
            pre, cur = cur, max(cur, pre + nums[i - 1])
        return cur
```

打家劫舍 2

213. 打家劫舍 II

难度 中等 292 收藏 分享 切换为英文 关注 反馈

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

示例 1：

```
输入: [2, 3, 2]
输出: 3
解释: 你不能先偷窃 1 号房屋 (金额 = 2), 然后偷窃 3 号房屋 (金额 = 2), 因为他们是相邻的。
```

示例 2：

```
输入: [1, 2, 3, 1]
输出: 4
解释: 你可以先偷窃 1 号房屋 (金额 = 1), 然后偷窃 3 号房屋 (金额 = 3)。
偷窃到的最高金额 = 1 + 3 = 4。
```

1、# 第一间房屋必偷，偷到第 N-1 间房屋为止

2、# 第一间不偷，偷到最后一间房屋为止

```
class Solution:
    def rob(self, nums: [int]) -> int:
        N = len(nums)
        return max(self.subRob(nums, 0, N - 2), self.subRob(nums, 1, N - 1))

    def subRob(self, nums, start, end): #从start 偷到 end处能获得的最高金额
        N = len(nums)
        pre, cur = 0, 0 # 因为从start才开始偷，所以都为0
        if N == 1:
            return nums[0]
        else:
            for i in range(start, end + 1):
                pre, cur = cur, max(cur, pre + nums[i])
        return cur
```

打家劫舍 3

337. 打家劫舍 III

难度 中等 383 收藏 分享 切换为英文 关注 反馈

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

输入: [3, 2, 3, null, 3, null, 1]



输出: 7

解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.

示例 2:

输入: [3, 4, 5, 1, 3, null, 1]



输出: 9

解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.

1、dfs 函数规划 : dfs(root) 返回偷 root 和不偷 root 最大金额

2、Base case : if root == None: return 0, 0

3、状态转移方程 :

```
Left, NoLeft = dfs(root.left)
Right, NoRight = dfs(root.right)
return root.val + NoLeft + NoRight, max(Left, NoLeft) + max(Right, NoRight)
```

注意：兄弟节点不算相邻，不偷的话，也还是可以选择偷子节点和不偷子节点。

```
class Solution:
    def rob(self, root: TreeNode) -> int:
        Y, N = self.subRob(root)
        return max(Y, N)

    def subRob(self, root): # 从root节点进入 偷root节点和不偷root节点返回的最大金额
        if root == None:
            return 0, 0
        leftY, leftN = self.subRob(root.left)
        rightY, rightN = self.subRob(root.right)
        Y = leftN + rightN + root.val
        N = max(leftY, leftN) + max(rightN, rightY) # 不偷的话，左右节点可选择偷或偷中较大的那个
        return Y, N
```

二维：

二维动态规划时的 base case 要注意当 $i==0$ 和 $j==0$ 时的 base case 是多少，而不是
单单只关注一个 $dp[0][0]$

最长回文子序列

516. 最长回文子序列

难度 中等 266 收藏 分享 切换为英文 关注 反馈

给定一个字符串 s ，找到其中最长的回文子序列，并返回该序列的长度。可以假设 s 的最大长度为 1000。

示例 1:

输入:

"bbbab"

输出:

4

一个可能的最长回文子序列为 "bbbb"。

示例 2:

输入:

"cbbd"

输出:

2

一个可能的最长回文子序列为 "bb"。

提示：

- $1 \leq s.length \leq 1000$
- s 只包含小写英文字母

Dp[i][j]表示从 s 的第 i 个字符到第 j 个字符包含的回文子序列的最大长度

```
class Solution:  
    def longestPalindromeSubseq(self, s: str) -> int:  
        N = len(s)  
        dp = [[0] * N for _ in range(N)]  
        for i in range(N - 1, -1, -1): # i从后往前, 当i = 0时, 得到最终结果  
            dp[i][i] = 1 # 必须在这里初始化, 并不是所有的dp[i][j]都等于1  
            for j in range(i + 1, N):  
                if s[i] == s[j]:  
                    dp[i][j] = dp[i + 1][j - 1] + 2  
                else:  
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])  
        return dp[0][N - 1]
```

最长公共子序列

1143. 最长公共子序列

难度 中等 159 收藏 分享 切换为英文 关注 反馈

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长公共子序列的长度。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。两个字符串的「公共子序列」是这两个字符串所共同拥有的子序列。

若这两个字符串没有公共子序列，则返回 0。

示例 1：

```
输入: text1 = "abcde", text2 = "ace"
输出: 3
解释: 最长公共子序列是 "ace"，它的长度为 3。
```

示例 2：

```
输入: text1 = "abc", text2 = "abc"
输出: 3
解释: 最长公共子序列是 "abc"，它的长度为 3。
```

示例 3：

```
输入: text1 = "abc", text2 = "def"
输出: 0
解释: 两个字符串没有公共子序列，返回 0。
```

提示:

- $1 \leq \text{text1.length} \leq 1000$
- $1 \leq \text{text2.length} \leq 1000$
- 输入的字符串只含有小写英文字母。

1、子问题规划： $\text{dp}[i][j]$ 表示 $s1[:i]$ 和 $s2[:j]$ 的最长公共子序列

2、Base case：均初始化为 0

3、状态转移方程： $\text{dp}[i][j] = \text{dp}[i-1][j-1] + 1 \quad \text{if } s1[i] == s2[j] \quad \text{else} \quad \max(\text{dp}[i-1][j], \text{dp}[i][j-1])$

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        M, N = len(text1), len(text2)
        dp = [[0] * (N + 1) for _ in range(M + 1)] # dp[i][j] 表示text1[:i]和text2[:j]的最长公共子序列
        for i in range(M):
            for j in range(N):
                if text1[i] == text2[j]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
        return dp[M - 1][N - 1]
```

编辑距离

72. 编辑距离

难度 困难 928 收藏 分享 切换为英文 关注 反馈

给你两个单词 $word_1$ 和 $word_2$ ，请你计算出将 $word_1$ 转换成 $word_2$ 所使用的最少操作数。

你可以对一个单词进行如下三种操作：

1. 插入一个字符
2. 删除一个字符
3. 替换一个字符

示例 1：

```
输入: word1 = "horse", word2 = "ros"
输出: 3
解释:
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

示例 2：

```
输入: word1 = "intention", word2 = "execution"
输出: 5
解释:
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')
```

1、子问题规划 : $dp[i][j]$ 表示 $word_1$ 前 i 个字符转化为 $word_2$ 前 j 个字符所使用的最少操作数

2、Base case：当 $i = 0$ 时： $dp[i][j] = j$ ； 当 $j = 0$ 时： $dp[i][j] = i$ ；其余初始化为 $M+N$

3、状态转移方程：

插入： $dp[i][j] = dp[i][j-1] + 1$

删除 : $dp[i][j] = dp[i-1][j] + 1$

替换 : $dp[i][j] = dp[i-1][j-1] + 1$ if $word1[i - 1] \neq word1[j - 1]$ else
 $dp[i - 1][j - 1]$
 $dp[i][j] = \text{上面最小值}$

看一个例子就懂 :

horse --> ros

插入 s: horse --> ro $dp[i][j-1]$

删除 e: hors --> ros $dp[i-1][j]$

替换 e: hors --> ro $dp[i-1][j-1]$

注 : 1、当比较的字母相等时 , 不需要替换操作

2、当 $word1$ 长度为 0 时 , 要插入 $word2$ 长度 , 反之删除 $word1$ 长度

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        M, N = len(word1), len(word2)
        dp = [[M+N] * (N+1) for _ in range(M+1)] # dp[i][j] 表示 word1 前 i 个字符到 word2 前 j 个字符的编辑距离
        for i in range(M+1):
            for j in range(N+1):
                if i == 0:
                    dp[i][j] = j
                elif j == 0:
                    dp[i][j] = i
                else:
                    delete = dp[i-1][j] + 1
                    insert = dp[i][j-1] + 1
                    replace = dp[i-1][j-1]
                    if word1[i-1] != word2[j-1]: # word1 第 i 个字符和 word2 第 j 个字符相等时, 相等的时候不需要替换操作
                        replace += 1
                    dp[i][j] = min(delete, insert, replace)
        return dp[M][N]
```

不同的子序列

115. 不同的子序列

难度 困难 206 喜欢 例 A 举一反三

给定一个字符串 **S** 和一个字符串 **T**，计算在 **S** 的子序列中 **T** 出现的个数。

一个字符串的一个子序列是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，“ACE”是“ABCDE”的一个子序列，而“AEC”不是）

题目数据保证答案符合 32 位带符号整数范围。

示例 1：

输入：S = "rabbbit", T = "rabbit"

输出：3

解释：

如下图所示，有 3 种可以从 S 中得到 "rabbit" 的方案。
(上箭头符号 ^ 表示选取的字母)

rabbbit

 ^ ^ ^ ^ ^ ^

rabbbit

 ^ ^ ^ ^ ^ ^

rabbbit

 ^ ^ ^ ^ ^ ^

示例 2:

输入: $s = "babgbag"$, $T = "bag"$

输出: 5

解释:

如下图所示，有 5 种可以从 s 中得到 "bag" 的方案。
(上箭头符号 ^ 表示选取的字母)

```
babgbag
^ ^ ^
babgbag
^ ^   ^
babgbag
^   ^ ^
babgbag
^   ^ ^
babgbag
    ^ ^
babgbag
      ^ ^
```

1、子问题规划 : $dp[i][j]$ 表示 S 前 i 个字符含 T 前 j 个字符的方案数

2、Base case : $dp = [[0] * N \text{ for } _ \text{in range}(M)]$, $dp[i][0] = 1$, 当 T 为空时，永远都是 1 种组成

3、状态转移方程：

$$dp[i][j] = dp[i-1][j] \text{ if } S[i-1] \neq T[i-1] \text{ else } dp[i-1][j] + dp[i-1][j-1]$$

注解：当不相等时： $dp[i][j] = dp[i - 1][j]$

当相等时： $dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1]$

```

class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        n1 = len(s)
        n2 = len(t)
        dp = [[0] * (n2 + 1) for _ in range(n1 + 1)]
        for i in range(n1 + 1):
            dp[i][0] = 1
        for i in range(1, n1 + 1):
            for j in range(1, n2 + 1):
                if s[i - 1] == t[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]
                else:
                    dp[i][j] = dp[i - 1][j]
        return dp[-1][-1]

```

数字三角形

120. 三角形最小路径和

难度 中等 426 收藏 分享 贡献

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。

例如，给定三角形：

```
[
    [2],
    [3,4],
    [6,5,7],
    [4,1,8,3]
]
```

自顶向下的最小路径和为 11 (即, $2 + 3 + 5 + 1 = 11$)。

1、子问题规划： $dp[i][j]$ 表示移动该点的最小路径和

2、Base case： $dp[0][0] = triangle[0][0]$

3、状态转移方程：

```

if i == 0:
    dp[i][j] = dp[i-1][j] + triangle[i][j]
elif i == len(triangle[i]) - 1:
    dp[i][j] = dp[i-1][j-1] + triangle[i][j]
else:
    dp[i][j] = min(dp[i-1][j], dp[i-1][j-1]) + triangle[i][j]
return min(dp[len(triangle)-1])

```

我的思路：从顶点到底部，路径有很多条，但是起点只有一个，终点只有最后一行数字，

那么建立一个数组 $dp[i][j]$ 代表的是从起点到达该点的最短路径。 $dp[i][j]$ 又分为三种类型：

左边缘点，中间点和右边缘点。

```

class Solution:
    def minimumTotal(self, triangle):
        rowNum = len(triangle)
        for i in range(1, rowNum):
            for j in range(len(triangle[i])):
                if j == 0:
                    triangle[i][j] += triangle[i - 1][j]
                elif j == len(triangle[i]) - 1:
                    triangle[i][j] += triangle[i - 1][j - 1]
                else:
                    triangle[i][j] += min(triangle[i - 1][j], triangle[i - 1][j - 1])
        return min(triangle[rowNum - 1])

```

不同的路径 1

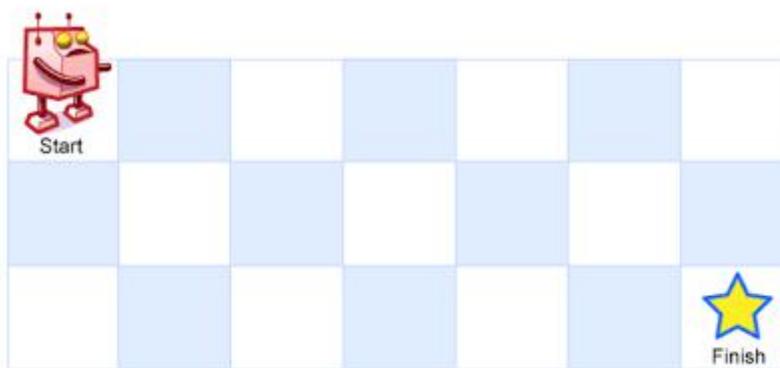
62. 不同路径

难度 中等 575

一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径？



例如，上图是一个 7×3 的网格。有多少可能的路径？

1、子问题规划： $dp[i][j]$ 表示到达 (i, j) 这个位置的不同的路径数

2、Base case： $dp[0][0] = 1$, 其余均为 0

3、状态转移方程：

```
If i == 0:  
    dp[i][j] = dp[i][j-1]  
elif j == 0:  
    dp[i][j] = dp[i-1][j]  
  
else :  
    dp[i][j] = dp[i-1][j] + dp[i][j-1]
```

解题思路：因为只有往右往下两个方向，那么到二维网格上的每个点的路径条数都等于上一个路径点的路径条数加上左边一个路径点的路径条数之和，增加一行一列用作第一行第一列之用。能用该种方法的原因是因为只有两个方向，那么所有的路径都不会重叠。

```

class Solution:
    def uniquePaths(self, m, n):
        mp = [[0] * n for _ in range(m)]
        for i in range(m):
            for j in range(n):
                if (i == 0 or j == 0): # 第一行和第一列都只有一条路径可达
                    mp[i][j] = 1
                else:
                    mp[i][j] = mp[i - 1][j] + mp[i][j - 1]
        return mp[m - 1][n - 1]

```

不同的路径 2

63. 不同路径 II

难度 中等 306 收藏 分享

一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？



网格中的障碍物和空位置分别用 1 和 0 来表示。

说明：m 和 n 的值均不超过 100。

我的思路：相比于上一题，其实还是多了障碍物，其实还是只有两个方向，即向右和向下，那么其实还是一道动态规划问题，如果节点的上面或者左边有障碍物的话路径数清 0.

```

class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid):
        m, n = len(obstacleGrid), len(obstacleGrid[0])
        mp = [[0] * n for _ in range(m)]
        mp[0][0] = 1
        for i in range(m):
            for j in range(n):
                if obstacleGrid[i][j] == 1: # 遇到障碍物，路径清0
                    mp[i][j] = 0
                elif i == 0 and j == 0: # 初始化
                    mp[i][j] = 1
                elif i == 0: # 三种情况讨论
                    mp[i][j] = mp[i][j - 1]
                elif j == 0:
                    mp[i][j] = mp[i - 1][j]
                else:
                    mp[i][j] = mp[i - 1][j] + mp[i][j - 1]
        return mp[m - 1][n - 1]

```

待：不同的路径 3

980. 不同路径 III

难度 困难 山 68 ❤️ ⚡ 🌐 🔍

在二维网格 grid 上，有 4 种类型的方格：

- 1 表示起始方格。且只有一个起始方格。
- 2 表示结束方格，且只有一个结束方格。
- 0 表示我们可以走过的空方格。
- -1 表示我们无法跨过的障碍。

返回在四个方向（上、下、左、右）上行走时，从起始方格到结束方格的不同路径的数目，每一个无障碍方格都要通过一次。

示例 1：

```

输入： [[1,0,0,0],[0,0,0,0],[0,0,2,-1]]
输出： 2
解释： 我们有以下两条路径：
1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),
(2,0),(2,1),(2,2)
2. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),
(1,3),(1,2),(2,2)

```

示例 2:

输入: [[1,0,0,0],[0,0,0,0],[0,0,0,2]]

输出: 4

解释: 我们有以下四条路径:

1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2),(2,3)
2. (0,0),(0,1),(1,1),(1,0),(2,0),(2,1),(2,2),(1,2),(0,2),(0,3),(1,3),(2,3)
3. (0,0),(1,0),(2,0),(2,1),(2,2),(1,2),(1,1),(0,1),(0,2),(0,3),(1,3),(2,3)
4. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2),(2,3)

示例 3:

输入: [[0,1],[2,0]]

输出: 0

解释:

没有一条路能完全穿过每一个空的方格一次。

请注意，起始和结束方格可以位于网格中的任意位置。

骰子求和

剑指 Offer 60. n个骰子的点数

难度 简单 70 收藏 分享 切换为英文 关注 反馈

把n个骰子扔在地上，所有骰子朝上一面的点数之和为s。输入n，打印出s的所有可能的值出现的概率。

你需要用一个浮点数数组返回答案，其中第 i 个元素代表这 n 个骰子所能掷出的点数集合中第 i 小的那个的概率。

示例 1:

输入: 1

输出: [0.16667,0.16667,0.16667,0.16667,0.16667,0.16667]

示例 2:

输入: 2

输出: [0.02778,0.05556,0.08333,0.11111,0.13889,0.16667,0.13889,0.11111,0.08333,0.05556,0.02778]

1、子问题规划：设 $dp[i][j]$ ，表示 i 个骰子，骰出和为 j 的概率。

2、base case : $dp = [[0] * (6*n + 1) \text{ for } _ \text{in} \text{ range}(n+1)]$, $dp[1][1\sim 6] = 1/6$

3、状态转移方程为：

```
for i in range(1, n+1):
    for j in range(i, 6 * i + 1):
        for k in (1, 7):
            if j - k >= 0
                dp[i][j] += dp[i-1][j-k]
            dp[i][j] /= 6
return dp[-1][n:]
```

4、空间优化：不建议

```
class Solution:
    def twoSum(self, n):
        ans = []
        f = [[0 for j in range(6 * n + 1)] for j in range(n + 1)]
        for i in range(1, 7):
            f[1][i] = 1.0 / 6.0 # 在python中其实除数为不为浮点数，没关系，因为单个斜杠代表求浮点数商
        for i in range(2, n + 1): # 从第二个骰子开始
            for j in range(i, 6 * n + 1): # 多一个骰子多出六种情况
                for k in range(1, 7):
                    if j - k > 0: # 当然不能为负数
                        f[i][j] += f[i - 1][j - k]
            f[i][j] /= 6 # 按理来说应该除以6*6，但是前面每个阶段都除以了一个6，所以这里只需要除以一个6
        for i in range(n, 6 * n + 1):
            ans.append(f[n][i])
        return ans
```

戳气球

312. 戳气球

难度 困难 340 收藏 分享 切换为英文 关注 反馈

有 n 个气球，编号为 0 到 $n-1$ ，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。每当你戳破一个气球 i 时，你可以获得 $\text{nums}[\text{left}] * \text{nums}[i] * \text{nums}[\text{right}]$ 个硬币。这里的 left 和 right 代表和 i 相邻的两个气球的序号。注意当你戳破了气球 i 后，气球 left 和气球 right 就变成了相邻的气球。

求所能获得硬币的最大数量。

说明:

- 你可以假设 $\text{nums}[-1] = \text{nums}[n] = 1$ ，但注意它们不是真实存在的所以并不能被戳破。
- $0 \leq n \leq 500, 0 \leq \text{nums}[i] \leq 100$

示例:

```
输入: [3,1,5,8]
输出: 167
解释: nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
      coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167
```

我的思路：又是求最值，那肯定要用动态规划， $\text{dp}[i][j]$ 表示吹爆所有在区间 $[i,j]$ 的气球，所

能得到的最大分数。那么如何进行状态转移？假设吹爆第 k 个气球，那么 $k-1$ 和 $k+1$ 个气球变得相邻，不方便转移。那么我们就先吹爆区间 $[i, k-1]$ 和区间 $[k+1, j]$ 的所有气球，最后再来吹爆第 k 个气球，那么这样就十分方便转移了。枚举 k 的位置，得到状态转移方程为：

$$\begin{aligned} dp(i,j) &= \max\{dp(i,k-1) + score(k) + dp(k+1,j) | (i \leq k \leq j)\}, \\ score(k) &= a[i-1] * a[k] * a[r+1] \end{aligned}$$

1、子问题规划： $dp[i][j]$ 表示戳爆区间 $[i][j]$ 区间的气球所能得到的最大分数

2、Base case： $dp[0][0] = nums[0] + nums[1]$ ，求最大值，初始化为 0

3、状态转移方程： $dp[i][j] = \max(dp[i][k - 1] + score[k] + dp[k + 1][j])$

$$Score[k] = nums[i] + nums[k] + nums[j] // ??$$

```
class Solution:
    def maxCoins(self, nums):
        if not nums:
            return 0
        nums = [1, *nums, 1] # 在数组nums两边补1
        n = len(nums)
        dp = [[0] * n for _ in range(n)]
        for i in range(n - 1, -1, -1): # 左指针
            for j in range(i + 2, n): # 右指针
                for k in range(i + 1, j): # 穿爆中间的气球，两端不可取
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + nums[i] * nums[k] * nums[j])
        return dp[0][n - 1]
```

非最值问题

单词拆分

139. 单词拆分

难度 中等 552 收藏 分享 切换为英文 关注 反馈

给定一个非空字符串 s 和一个包含非空单词列表的字典 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

示例 2：

输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
注意你可以重复使用字典中的单词。

示例 3：

输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false

1、子问题规划：dp[n]表示字符串的前 n 个字符是否合法

2、Base case： dp[0] = True

3、状态转移方程：dp[n] = dp[j] && check[j, n] break

```
class Solution:  
    def wordBreak(self, s: str, wordDict: [str]) -> bool:  
        N = len(s)  
        dp = [False] * (N + 1)  
        dp[0] = True  
        for i in range(N + 1):  
            for j in range(i + 1):  
                cur = s[j:i]  
                if dp[j] and cur in wordDict:  
                    dp[i] = True  
                    break  
        return dp[-1]
```

单词拆分 2

140. 单词拆分 II

难度 困难 185 收藏 分享 切换为英文 关注 反馈

给定一个非空字符串 s 和一个包含非空单词列表的字典 $wordDict$, 在字符串中增加空格来构建一个句子, 使得句子中所有的单词都在词典中。返回所有这些可能的句子。

说明:

- 分隔时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1:

```
输入:  
s = "catsanddog"  
wordDict = ["cat", "cats", "and", "sand", "dog"]  
输出:  
[  
    "cats and dog",  
    "cat sand dog"  
]
```

示例 2:

```
输入:  
s = "pineapplepenapple"  
wordDict = ["apple", "pen", "applepen", "pine", "pineapple"]  
输出:  
[  
    "pine apple pen apple",  
    "pineapple pen apple",  
    "pine applepen apple"  
]  
解释: 注意你可以重复使用字典中的单词。
```

示例 3:

```
输入:  
s = "catsandog"  
wordDict = ["cats", "dog", "sand", "and", "cat"]  
输出:  
[]
```

1、子问题规划 : $dp[n]$ 表示 s 的前 n 个字符包含的所有可能的句子

2、Base case : $dp = [[] \text{ for } _ \text{ in } \text{range}(N)]$, $dp[0] = ["]$

3、状态转移方程 :

```
For i in range(1, N + 1):  
    List = []  
    for j in range(j):  
        Cur = s[j:i]  
        if check(cur):
```

```

        for x in dp[j]:
            midSymbol = "" if len(x) == 0 else " "
            List.append(x + midSymbol + cur)
    dp[i] = list
Return dp[-1]

```

```

class Solution:
    def wordBreak(self, s: str, wordDict: [str]) -> bool:
        N = len(s)
        dp = [[] for _ in range(N + 1)]
        dp[0].append("1")
        for i in range(N + 1):
            list = [] # 空字符串
            for j in range(i + 1):
                cur = s[j:i]
                if len(dp[j]) != 0 and cur in wordDict:
                    for x in dp[j]:
                        midSymbol = "" if len(x) == 0 else " "
                        list.append(x + midSymbol + cur)
            dp[i] = list if len(list) != 0 else dp[i]
        # print(dp)
        result = []
        for x in dp[-1]:
            result.append(x[2:-1])
        return result

```

```

[[['1'], [], [], [], ['1 leet'], [], [], [], ['1 leet code']]]
['leet code']

```

```

class Solution:
    def wordBreak(self, s, wordDict):
        List = []
        self.dfs2(s, wordDict, "", List)
        return List

    def dfs2(self, s, wordDict, string, List):
        if len(s) == 0:
            List.append(string.strip())
            return
        for i in range(1, len(s) + 1):
            prefix = s[:i]
            if prefix not in wordDict:
                continue
            self.dfs2(s[i:], wordDict, string + " " + prefix, List)

```

都超出时间限制

我觉得这种求方案数所有情况的应该用回溯，如果问方案种数可以用 dp

贪心

买卖股票的最佳时机 1

121. 买卖股票的最佳时机

难度 简单 1024 收藏 分享 切换为英文 关注 反馈

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

示例 1：

输入： [7,1,5,3,6,4]

输出： 5

解释： 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5。

注意利润不能是 7-1 = 6，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

示例 2：

输入： [7,6,4,3,1]

输出： 0

解释： 在这种情况下，没有交易完成，所以最大利润为 0。

```
import sys
class Solution:
    def maxProfit(self, prices: [int]) -> int:
        maxProfit, minVal = 0, sys.maxsize
        for x in prices:
            minVal = min(minVal, x) # 维持一个最小值是关键
            maxProfit = max(maxProfit, x - minVal)
        return maxProfit
```

买卖股票的最佳时机 2

122. 买卖股票的最佳时机 II

难度 简单 744 收藏 分享 切换为英文 关注 反馈

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

输入：[7, 1, 5, 3, 6, 4]

输出：7

解释：在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5 - 1 = 4$ 。

随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = $6 - 3 = 3$ 。

示例 2：

输入：[1, 2, 3, 4, 5]

输出：4

解释：在第 1 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5 - 1 = 4$ 。

注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。

因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

示例 3：

输入：[7, 6, 4, 3, 1]

输出：0

解释：在这种情况下，没有交易完成，所以最大利润为 0。

```
class Solution:
    def maxProfit(self, prices: [int]) -> int:
        low, high = 0, 0 # 买入
        N, sum = len(prices), 0
        for i in range(1, N):
            if prices[i] >= prices[high]:
                high = i # 股票上升趋势，往后接着看
            else:
                sum += prices[high] - prices[low] # 股票开始下降，抛出
                low, high = i, i # 重新买入
        return prices[high] - prices[low] + sum # 股市关闭，强制卖掉
```

买卖股票的最佳时机 3

123. 买卖股票的最佳时机 III

难度 困难 417 收藏 分享 切换为英文 关注 反馈

给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 两笔 交易。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

输入： [3, 3, 5, 0, 0, 3, 1, 4]

输出： 6

解释： 在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润 = $3 - 0 = 3$ 。

随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天（股票价格 = 4）的时候卖出，这笔交易所能获得利润 = $4 - 1 = 3$ 。

示例 2：

输入： [1, 2, 3, 4, 5]

输出： 4

解释： 在第 1 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5 - 1 = 4$ 。

注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。

因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

示例 3：

输入： [7, 6, 4, 3, 1]

输出： 0

解释： 在这个情况下，没有交易完成，所以最大利润为 0。

1、思路一：(leetcode 上除了最后一个测试用例超时之外，其余都能通过)

```
import sys

class Solution:
    def maxProfit2(self, prices: [int]) -> int:
        maxProfit, N = 0, len(prices)
        for j in range(N): # 遍历中间点，将数组分为两半，分别求最大利润
            leftMin, leftMaxProfit = sys.maxsize, 0
            rightMax, rightMaxProfit = 0, 0
            for i in range(j + 1):
                leftMin = min(leftMin, prices[i])
                leftMaxProfit = max(leftMaxProfit, prices[i] - leftMin)
            for k in range(N - 1, j - 1, -1):
                rightMax = max(rightMax, prices[k])
                rightMaxProfit = max(rightMaxProfit, rightMax - prices[k])
            maxProfit = max(maxProfit, leftMaxProfit + rightMaxProfit)
        return maxProfit
```

2、思路二：状态机 DP

两次买卖，可以分为五种状态：

S0：可交易两次，不持股，可买入

S1：可交易一次，持股，可卖出

S2：可交易一次，不持股，可买入

S3：可交易 0 次，持股，可卖出

S4：可交易 0 次，不持股，可买入



子问题间的状态转移关系（状态机）

五种状态，可选择保持状态，也可进入下一个状态，用 max 函数来进行选择保持

还是进入下一个状态。 s_0 始终为 0，不参与进来。

```
for (int k = 1; k <= n; k++) {  
    s1[k] = Math.max(s1[k-1], -p[k-1]);  
    s2[k] = Math.max(s2[k-1], s1[k-1] + p[k-1]);  
    s3[k] = Math.max(s3[k-1], s2[k-1] - p[k-1]);  
    s4[k] = Math.max(s4[k-1], s3[k-1] + p[k-1]);  
}  
return Math.max(0, Math.max(s2[n], s4[n]));
```

空间进行优化后代码为：

```
public int maxProfit(int[] prices) {
    if (prices.length == 0) {
        return 0;
    }

    int s1 = Integer.MIN_VALUE;
    int s2 = 0;
    int s3 = Integer.MIN_VALUE;
    int s4 = 0;

    for (int p : prices) {
        s4 = Math.max(s4, s3 + p);
        s3 = Math.max(s3, s2 - p);
        s2 = Math.max(s2, s1 + p);
        s1 = Math.max(s1, -p);
    }
    return Math.max(0, Math.max(s2, s4));
}
```

因为 `s1` 和 `s3` 持有股票，不能买入，故强制卖出，所以将其设为整数最小值。

跳跃游戏

55. 跳跃游戏

难度 中等 710 收藏 叉

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1：

输入： [2,3,1,1,4]

输出： true

解释：我们可以先跳 1 步，从位置 0 到达 位置 1，然后再从位置 1 跳 3 步到达最后一个位置。

示例 2：

输入： [3,2,1,0,4]

输出： false

解释：无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0，所以你永远不可能到达最后一个位置。

贪心思想：只关注最远距离

关键：维持一个最远距离

```
class Solution:
    def canJump(self, A):
        N = len(A)
        maxPos = 0
        for i in range(N):
            if maxPos >= i and i + A[i] > maxPos:
                maxPos = i + A[i] # 如果当前位置可达，并且从此位置起能够达到的距离大于最远距离
        return maxPos >= N - 1
```

待：跳跃游戏 2

45. 跳跃游戏 II

难度 困难 601 喜欢 例题 文档

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例：

输入： [2,3,1,1,4]

输出： 2

解释：跳到最后一个位置的最小跳跃数是 2。

从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

说明：

假设你总是可以到达数组的最后一个位置。

1、子问题规划： $dp[n]$ 表示跳到 n 这个位置最少需要多少步

2、Base case： $dp[0]=0$, $dp[1]=1$, $dp[n] = sys.maxsize$

3、状态转移方程： $dp[i] = \min(dp[i], dp[j] + 1)$ if $nums[j] \geq i - j$

```
class Solution:
    def jump(self, nums: [int]) -> int:
        N = len(nums)
        if N == 0:
            return 0
        if N == 1: # 本身就在最后一个位置，不需要跳
            return 0
        dp = [sys.maxsize] * N
        dp[0], dp[1] = 0, 1
        for i in range(2, N):
            for j in range(i): # 遍历上一步可能的位置
                if nums[j] >= i - j: # 从该位置能跳到第i个位置
                    dp[i] = min(dp[i], dp[j] + 1)
        return dp[-1]
```

采取上述方法会超时，所以还是要用贪心的思想解题。

下一跳能跳的最远的位置 maxPos , end 为下一跳最晚起跳的位置 , 步数 step。

- 1、maxPos = nums[0], end = maxPos, step = 1
- 2、For i in range(1, N):
 If maxPos > i:
 maxPos = max(maxPos, i + nums[i])
 If i == end:
 Step += 1
 end = maxPos

```
class Solution:  
    def jump(self, nums: [int]) -> int:  
        n = len(nums)  
        maxPos, end, step = 0, 0, 0 # end记录最晚起跳的位置  
        for i in range(n - 1): # 到了倒数第2个位置，无论如何都是要起跳的，肯定能到最后一个位置。到n的话会多跳一步  
            if maxPos >= i: # 能跳到该位置  
                maxPos = max(maxPos, i + nums[i])  
            if i == end: # 到了最晚起跳的位置，该起跳了  
                end = maxPos  
                step += 1  
        return step
```

空间复杂度

子矩阵和为 0

1074. 元素和为目标值的子矩阵数量

难度 困难 点赞 41 收藏 分享 切换为英文 关注 反馈

给出矩阵 matrix 和目标值 target，返回元素总和等于目标值的非空子矩阵的数量。

子矩阵 x_1, y_1, x_2, y_2 是满足 $x_1 \leq x \leq x_2$ 且 $y_1 \leq y \leq y_2$ 的所有单元 $\text{matrix}[x][y]$ 的集合。

如果 (x_1, y_1, x_2, y_2) 和 (x'_1, y'_1, x'_2, y'_2) 两个子矩阵中部分坐标不同（如： $x_1 \neq x'_1$ ），那么这两个子矩阵也不同。

示例 1：

```
输入：matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0
输出：4
解释：四个只含 0 的 1x1 子矩阵。
```

示例 2：

```
输入：matrix = [[1,-1],[-1,1]], target = 0
输出：5
解释：两个 1x2 子矩阵，加上两个 2x1 子矩阵，再加上一个 2x2 子矩阵。
```

提示：

1. $1 \leq \text{matrix.length} \leq 300$
2. $1 \leq \text{matrix[0].length} \leq 300$
3. $-1000 \leq \text{matrix[i]} \leq 1000$
4. $-10^8 \leq \text{target} \leq 10^8$

求矩阵和：

1、子问题规划： $\text{sum}[i][j]$ 表示以 $\text{matrix}[i][j]$ 结尾的子矩阵和

2、Base case：

```
if i == 0:  
    sum[i][j] = sum[i][j-1] + matrix[i][j]  
elif j == 0:  
    sum[i][j] = sum[i-1][j] + matrix[i][j]
```

3、状态转移：

求矩阵和：

```
sum[i][j] = sum[i][j-1]+sum[i-1][j]-sum[i-1][j-1]+matrix[i][j]
```

求子矩阵和：

```
subSum(m, n) --> (x, y) = sum[x][y] - sum[m-1][y] - sum[x][n-1] + sum[m-1][n-1]
```

```

class Solution:
    def numSubmatrixSumTarget(self, matrix, target):
        row, col = len(matrix), len(matrix[0])
        prefixSum = [[0] * (col + 1) for _ in range(row + 1)]
        count = 0
        for i in range(1, row + 1):
            for j in range(1, col + 1):
                prefixSum[i][j] = prefixSum[i - 1][j] + prefixSum[i][j - 1] - prefixSum[i - 1][j - 1] + matrix[i - 1][j - 1]
                for x in range(i): # 这里求的是第一个点的坐标，在第二个点的坐标内
                    for y in range(j):
                        key = target - prefixSum[i][j] + prefixSum[i][y] + prefixSum[x][j]
                        if key == prefixSum[x][y]: # 比较的是以第一个坐标为右下角的矩阵和
                            count += 1
        return count

```

Case 37/38

数组中重复的数据

442. 数组中重复的数据

难度 中等 236 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 a , 其中 $1 \leq a[i] \leq n$ (n 为数组长度), 其中有些元素出现两次而其他元素出现一次。

找到所有出现两次的元素。

你可以不用到任何额外空间并在 $O(n)$ 时间复杂度内解决这个问题吗?

示例：

输入:

[4, 3, 2, 7, 8, 2, 3, 1]

输出:

[2, 3]

```

class Solution:
    def findDuplicates(self, nums: [int]) -> [int]:
        N, res = len(nums), []
        for num in nums:
            index = abs(num) - 1 # 因为nums里的值是1-n
            if nums[index] < 0:
                res.append(index + 1)
            nums[index] = -abs(nums[index]) # 防止负负得正
        return res

```

找到数组中所有消失的数字

448. 找到所有数组中消失的数字

难度 简单 409 收藏 分享 切换为英文 关注 反馈

给定一个范围在 $1 \leq a[i] \leq n$ ($n = \text{数组大小}$) 的整型数组，数组中的元素一些出现了两次，另一些只出现一次。

找到所有在 $[1, n]$ 范围之间没有出现在数组中的数字。

您能在不使用额外空间且时间复杂度为 $O(n)$ 的情况下完成这个任务吗？你可以假定返回的数组不算在额外空间内。

示例：

输入：

`[4, 3, 2, 7, 8, 2, 3, 1]`

输出：

`[5, 6]`

```
class Solution:
    def findDisappearedNumbers(self, nums: [int]) -> [int]:
        N, res = len(nums), []
        if N == 0:
            return []
        for i in range(N):
            index = abs(nums[i])
            nums[index - 1] = -abs(nums[index - 1])
        for i in range(N):
            if nums[i] > 0:
                res.append(i + 1)
        return res
```

缺失的第一个正数

41. 缺失的第一个正数

难度 困难 | 688 | 收藏 | 分享 | 切换为英文 | 关注 | 反馈

给你一个未排序的整数数组，请你找出其中没有出现的最小的正整数。

示例 1：

输入： [1, 2, 0]
输出： 3

示例 2：

输入： [3, 4, -1, 1]
输出： 2

示例 3：

输入： [7, 8, 9, 11, 12]
输出： 1

提示：

你的算法的时间复杂度应为 $O(n)$ ，并且只能使用常数级别的额外空间。

肯定是要用到 hash 表的，但是这个 hash 表需要重用原表。

明白一个道理：缺失的数肯定在 $[1, \text{len}(\text{nums}) + 1]$ 范围内。

```
class Solution:
    def firstMissingPositive(self, nums: [int]) -> int:
        N = len(nums)
        for i in range(N):
            if nums[i] <= 0:
                nums[i] = N + 1 # 因为要利用负数进行打标记

        for i in range(N):
            index = abs(nums[i]) - 1
            if index < N:
                nums[index] = -abs(nums[index])

        for i in range(N):
            if 0 < nums[i]:
                return i + 1

        return N + 1
```

旋转数组

189. 旋转数组

难度 简单 632 收藏 分享 切换为英文 关注 反馈

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

示例 1：

```
输入: [1,2,3,4,5,6,7] 和 k = 3
输出: [5,6,7,1,2,3,4]
解释:
向右旋转 1 步: [7,1,2,3,4,5,6]
向右旋转 2 步: [6,7,1,2,3,4,5]
向右旋转 3 步: [5,6,7,1,2,3,4]
```

示例 2：

```
输入: [-1,-100,3,99] 和 k = 2
输出: [3,99,-1,-100]
解释:
向右旋转 1 步: [99,-1,-100,3]
向右旋转 2 步: [3,99,-1,-100]
```

说明：

- 尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。
- 要求使用空间复杂度为 $O(1)$ 的原地算法。

```
class Solution:
    def rotate(self, nums: [int], k: int) -> None:
        k = k % len(nums) # k可大于数组长度
        self.reverseList(nums, 0, len(nums) - 1)
        self.reverseList(nums, 0, k - 1)
        self.reverseList(nums, k, len(nums) - 1)

    def reverseList(self, List, start, end):
        while start < end:
            List[start], List[end] = List[end], List[start]
            start += 1
            end -= 1
```

旋转图像

48. 旋转图像

难度 中等 527 收藏 分享 切换为英文 关注 反馈

给定一个 $n \times n$ 的二维矩阵表示一个图像。

将图像顺时针旋转 90 度。

说明：

你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

示例 1：

```
给定 matrix =
[
    [1,2,3],
    [4,5,6],
    [7,8,9]
],  
  
原地旋转输入矩阵，使其变为：  
[  
    [7,4,1],  
    [8,5,2],  
    [9,6,3]  
]
```

示例 2：

```
给定 matrix =
[
    [ 5, 1, 9,11],
    [ 2, 4, 8,10],
    [13, 3, 6, 7],
    [15,14,12,16]
],  
  
原地旋转输入矩阵，使其变为：  
[  
    [15,13, 2, 5],  
    [14, 3, 4, 1],  
    [12, 6, 8, 9],  
    [16, 7,10,11]
]
```

```
class Solution(object):
    def rotate(self, matrix):
        n = len(matrix)
        for i in range(n // 2): # 把矩阵分成四块,注意下面是(n + 1) // 2, 看第一个例子可得原因
            for j in range((n + 1) // 2): # 神来之笔, 奇偶通吃
                temp = matrix[i][j] # 第一个元素
                matrix[i][j] = matrix[n - 1 - j][i] # 举例可得规律, 注意, 横纵坐标没有想等的
                matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j]
                matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i]
                matrix[j][n - 1 - i] = temp
```

下一个排列

31. 下一个排列

难度 中等 564 收藏 分享 切换为英文 关注 反馈

实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须原地修改，只允许使用额外常数空间。

以下是一些例子，输入位于左侧列，其相应输出位于右侧列。

1,2,3	→	1,3,2
3,2,1	→	1,2,3
1,1,5	→	1,5,1

思路：从后往前遍历，如果一直增大，那么就算换也无用，直到遇到下降的那个元素 a，那么就可以用后面的序列中的大于 a 的去替换 a，但是这个时候又需要保证是下一个排列，这个时候，后面的序列应该升序排列，保证最小，然后找到那个大于下降的元素，进行交换即可。

如：1,4,2,1 → 2,1,1,4 # 不能直接将 1 和 4 交换，4121 不是下一个排列。

```
class Solution:
    def nextPermutation(self, nums: [int]) -> None:
        N, flag = len(nums), False
        for i in range(N - 1, 0, -1):
            if nums[i - 1] < nums[i]: # 找到前面要替换的那个元素位置
                nums[i:] = sorted(nums[i:]) # 后面元素升序，使得结果尽可能小
                for j in range(i, N):
                    if nums[i - 1] < nums[j]: # 找到后面最小的那个元素位置
                        nums[i - 1], nums[j] = nums[j], nums[i - 1]
                        flag = True
                        break
                if flag:
                    break
        if not flag:
            nums.sort()
```

下一个更大的元素 III

556. 下一个更大元素 III

难度 中等 77 收藏 分享 切换为英文 关注 反馈

给定一个32位正整数 n，你需要找到最小的32位整数，其与 n 中存在的位数完全相同，并且其值大于n。如果不存在这样的32位整数，则返回-1。

示例 1：

输入: 12
输出: 21

示例 2：

输入: 21
输出: -1

注意：最大数不能超过 `math.pow(2, 32) // 2`

```

class Solution:
    def nextGreaterElement(self, n: int) -> int:
        strN = str(n)
        nums = [int(x) for x in strN]
        N = len(nums)
        for i in range(N - 1, 0, -1):
            if nums[i - 1] < nums[i]:
                nums[i:] = sorted(nums[i:])
                for j in range(i, N):
                    if nums[i - 1] < nums[j]:
                        nums[i - 1], nums[j] = nums[j], nums[i - 1]
                        result = int("".join(str(x) for x in nums))
                        if result >= math.pow(2, 32) // 2: # 1999999999 -> 9199999999
                            return -1
                        else:
                            return result
        return -1

```

移动 0 问题

283. 移动零

难度 简单 695 收藏 分享 切换为英文 关注 反馈

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

输入: [0,1,0,3,12]
输出: [1,3,12,0,0]

说明:

1. 必须在原数组上操作，不能拷贝额外的数组。
2. 尽量减少操作次数。

给定一个数组,本例实现将 `0` 移动到数组的最后面,非 `0` 元素保持原数组的顺序。注意必须在原数组上操作,最小化操作数。

解题思路:采用逆向思维,将所有的非 `0` 的数往前移动,直到找不到了 `0`,然后后面还剩几个位置,那么就补几个 `0`.

思路就是:`left` 指向非 `0` 数新位置,`right` 指向非 `0` 数的旧位置。

```

class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        left, right, N = 0, 0, len(nums)
        while right < N:
            if nums[right] != 0:
                nums[left] = nums[right]
                left += 1
            right += 1
        while left < N:
            nums[left] = 0
            left += 1

```

删除排序数组中的重复项

26. 删除排序数组中的重复项

难度 简单 收藏 分享 切换为英文 关注 反馈

给定一个排序数组，你需要在 **原地** 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 **原地** 修改输入数组 并在使用 $O(1)$ 额外空间的条件下完成。

示例 1:

给定数组 `nums = [1, 1, 2]`，
函数应该返回新的长度 **2**，并且原数组 `nums` 的前两个元素被修改为 **1, 2**。
你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 `nums = [0, 0, 1, 1, 1, 2, 2, 3, 3, 4]`，
函数应该返回新的长度 **5**，并且原数组 `nums` 的前五个元素被修改为 **0, 1, 2, 3, 4**。
你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢？

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下：

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
```

```
class Solution:
    def removeDuplicates(self, nums: [int]) -> int:
        left, right = 0, 0
        N = len(nums)
        while right < N:
            if nums[right] != nums[left]:
                left += 1 # left自增在前，保留重复项中的一位
                nums[left] = nums[right]
            right += 1
        return left + 1
```

移除元素

27. 移除元素

难度 简单 620 收藏 分享 切换为英文 关注 反馈

给你一个数组 $nums$ 和一个值 val , 你需要 原地 移除所有数值等于 val 的元素, 并返回移除后数组的新长度。

不要使用额外的数组空间, 你必须仅使用 $O(1)$ 额外空间并 原地 修改输入数组。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

示例 1:

```
给定 nums = [3, 2, 2, 3], val = 3,
函数应该返回新的长度 2，并且 nums 中的前两个元素均为 2。
你不需要考虑数组中超出新长度后面的元素。
```

```
class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:
        left, right, N = 0, 0, len(nums)
        while right < N:
            if nums[right] != val:
                nums[left] = nums[right]
                left += 1
            right += 1
        return left
```

数学问题

268. 缺失数字

难度 简单 300 收藏 分享 切换为英文 关注 反馈

给定一个包含 $0, 1, 2, \dots, n$ 中 n 个数的序列, 找出 $0..n$ 中没有出现在序列中的那个数。

示例 1:

```
输入: [3, 0, 1]
输出: 2
```

示例 2:

```
输入: [9, 6, 4, 2, 3, 5, 7, 0, 1]
输出: 8
```

1、利用求和公式：

```

class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        N = len(nums)
        Sum = (N + 1) * N // 2
        for num in nums:
            Sum -= num
        return Sum

class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        return sum(range(len(nums) + 1)) - sum(nums)

```

2、利用异或：

```

class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        res = 0
        for i in range(len(nums)):
            res ^= i ^ nums[i]
        return res ^ len(nums)

```

灯泡开关 I

319. 灯泡开关

难度 中等 123 收藏 分享 切换为英文 关注 反馈

初始时有 n 个灯泡关闭。第 1 轮，你打开所有的灯泡。第 2 轮，每两个灯泡你关闭一次。第 3 轮，每三个灯泡切换一次开关（如果关闭则开启，如果开启则关闭）。第 i 轮，每 i 个灯泡切换一次开关。对于第 n 轮，你只切换最后一个灯泡的开关。找出 n 轮后有多少个亮着的灯泡。

示例：

```

输入: 3
输出: 1
解释:
初始时, 灯泡状态 [关闭, 关闭, 关闭].
第一轮后, 灯泡状态 [开启, 开启, 开启].
第二轮后, 灯泡状态 [开启, 关闭, 开启].
第三轮后, 灯泡状态 [开启, 关闭, 关闭].

```

你应该返回 1，因为只有一个灯泡还亮着。

如果用模拟的话就会超时，如下：

```
class Solution:
    def bulbSwitch(self, n: int) -> int:
        arr = [0] * (n + 1) # arr[i]:第i个灯泡的状态
        for i in range(1, n + 1):
            for j in range(i, n + 1, i):
                arr[j] = 0 if arr[j] == 1 else 1
        import collections
        hash = collections.Counter(arr)
        return hash[1]
```

所以只能用数学问题解决：

假如该位置索引的因子有奇数个，那最后这个位置将会是亮着的，而一个数的因子是成对出现的，毫无疑问只有完全平方数的因子是奇数个，所以只需要记录灯牌所在位置索引有多少个完全平方数即可，而完全平方数和它的平方因子的是一一对应的，所以只需要求它的平方因子是多少，而平方因子又是一一增大的，所以只需要求出最大的那个平方因子是多少即可，最大平方因子是 $\text{int}(\sqrt{n})$

```
class Solution:
    def bulbSwitch(self, n: int) -> int:
        return int(n ** 0.5)
```

等差数列（华为）

题目描述

功能:等差数列 2, 5, 8, 11, 14。 . . .

输入:正整数N >0

输出:求等差数列前N项和

返回:转换成功返回 0 ,非法输入与异常返回-1

本题为多组输入，请使用while(cin>>)等形式读取数据

输入描述:

输入一个正整数。

输出描述:

输出一个相加后的整数。

示例1

输入	复制
2	
输出	复制
7	

```
while True:  
    try:  
        N = int(input())  
        cur, Sum = 2, 2  
        for _ in range(1, N):  
            cur += 3  
            Sum += cur  
        print(Sum)  
    except:  
        break
```

求解立方根（华为）

题目描述

• 计算一个数字的立方根，不使用库函数

详细描述：

• 接口说明

原型：

public static double getCubeRoot(double input)

输入:double 待求解参数

返回值:double 输入参数的立方根，保留一位小数

输入描述:

待求解参数 double类型

输出描述:

输入参数的立方根 也是double类型

示例1

输入	<input type="text" value="216"/> 复制
输出	<input type="text" value="6.0"/> 复制

加入可以使用库函数，那么可以这样 round(num**(1/3), 1)

```

def cubeRoot(N):
    left, right, neg = 0, abs(N), False
    if N < 0:
        neg = True
    interval = 0.001 # 取后两位，取后一位的话还是会出现四舍五入的情况
    while right - left > interval:
        mid = (left + right) / 2
        if mid * mid * mid < N:
            left = mid
        else:
            right = mid
    left = round(left, 1)
    print(left if not neg else -left)

while True:
    try:
        N = int(input())
        cubeRoot(N)
    except:
        break

```

求最小公倍数（华为）

题目描述

正整数A和正整数B 的最小公倍数是指 能被A和B整除的最小的正整数值，设计一个算法，求输入A和B的最小公倍数。

输入描述:

输入两个正整数A和B。

输出描述:

输出A和B的最小公倍数。

示例1

输入	复制
5 7	
输出	复制
35	

最小公倍数 = 两数之积除以最大公约数

```

while True:
    try:
        m, n = map(int, input().strip().split())
        product = m * n
        while n != 0:
            m, n = n, m % n
        print(product // m)
    except:
        break

```

求最大方差（大疆）

2. 给你一个数组，求一个k值，使得前k个数的方差+后面n-k个数的方差最小，并说明时间复杂度。

这道题容易想到的是暴力解法，从左往右遍历用数组记录左子数组均值，再从右往左遍历用数组记录右子数组均值，然后再遍历数组计算左、右子数组方差的和，时间复杂度O(n^2)，空间复杂度O(n)。

更快的解法：需要用到一个概念，方差等于平方的均值减均值的平方。利用这个公式，首先还是从左往右遍历数组，记录左子数组的方差，再从右往左遍历，记录右子数组的方差，最后遍历一次数组计算使方差和最小的k。时间复杂度O(n)，空间复杂度O(n)。

推导公式： $E[(x - u)^2] = E[x^2 - 2xu + u^2] = E(x^2) - u^2$ （E 代表求均值）

也就是：方差 = 各数平方的均值 - 各数均值的平方

```

if __name__ == '__main__':
    nums = [1, 1, 1, 3, 3]
    N = len(nums)
    left, right = [0] * N, [0] * N
    leftSum, rightSum = 0, 0
    leftSquareSum, rightSquareSum = 0, 0
    for i in range(N):
        leftSum += nums[i]
        leftSquareSum += nums[i] * nums[i]
        left[i] = leftSquareSum / (i + 1) - (leftSum / (i + 1)) * (leftSum / (i + 1)) # 各数平方和的均值 - 各数均值的平方

    nums = nums[::-1]
    for i in range(N):
        rightSum += nums[i]
        rightSquareSum += nums[i] * nums[i]
        right[i] = rightSquareSum / (i + 1) - (rightSum / (i + 1)) * (rightSum / (i + 1))

    k, Min = 0, sys.maxsize
    for i in range(N):
        if Min > left[i] + right[N - i - 2]:
            Min = left[i] + right[N - i - 2]
            k = i
    print(left)
    print(right)
    print(Min)
    print(k + 1)

```

完美数

507. 完美数

难度 简单 山 65 收藏 分享 切换为英文 关注 反馈

对于一个 **正整数**，如果它和除了它自身以外的所有正因子之和相等，我们称它为“完美数”。

给定一个 **整数 n**，如果他是完美数，返回 `True`，否则返回 `False`

示例：

输入: 28
输出: True
解释: $28 = 1 + 2 + 4 + 7 + 14$

一个数 N 与除了它自身之外的所有正因子之和相等称为完美数。

那就必须先求出其所有因子，我们可知，只需要求到 \sqrt{N} 之前的因子 $pres$ ，就能求出所有因子，因为 \sqrt{N} 之后的因子 $post$ 可以通过 $N // pre[i]$ 求出。

但这里要注意两点：1 是所有的数的因子，所以 sum 的初始值为 1，还有就是 \sqrt{N})

这个数，只能被加一次，所以要单独处理。

```
class Solution:
    def checkPerfectNumber(self, num: int) -> bool:
        sum = 1          # 因为除本身之外，所以1另算
        if num <= 0:
            return False
        import math
        sqrt = math.sqrt(num)
        for i in range(2, math.ceil(sqrt)): # 记得取整
            if num % i == 0:
                sum += i
                sum += num // i
        if sqrt * sqrt == num: # 因为只能加一次，所以另算
            sum += sqrt
        return sum == num
```

完美平方数

279. 完美平方数

难度 中等 469 收藏 分享 切换为英文 关注 反馈

给定正整数 n ，找到若干个完全平方数（比如 $1, 4, 9, 16, \dots$ ）使得它们的和等于 n 。你需要让组成和的完全平方数的个数最少。

示例 1：

```
输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4.
```

示例 2：

```
输入: n = 13
输出: 2
解释: 13 = 4 + 9.
```

因为一个数至多有 4 个完全平方数组成。

- 1、去除数中的 4,因为 4 的任何倍数都是完全平方数,无碍个数的多少
- 2、如果取 8 的余数是 7,那么就是 4 (1,1,1,2)
- 3、取 2 和取 1 能够判断出来
- 4、剩下的就是取 3.

```

class Solution:
    def numSquares(self, n):
        while n % 4 == 0: # 去除4的倍数
            n //= 4
        if n % 8 == 7:
            return 4
        for i in range(int(n ** 0.5) + 1): # 保证本身可以取到
            temp = i * i
            temp2 = n - i * i
            if int(temp2 ** 0.5) ** 2 == temp2: # 判断temp2是否是完全平方
                return 1 + (0 if temp == n or temp2 == n else 1) # 如果其中有一个正好等于n，那么就返回1，否则返回2
        return 3

```

找出 n 以内的素数（素数筛）

普通人的三部曲：

- 1、将偶数去掉
- 2、只需判断 $n \% \text{math.sqrt}(n) == 0$
- 3、数学定理：只有形如 $6n-1$ 和 $6n+1$ 的自然数可能是素数， $n \geq 1$ 。（所以 2,3 是例外）

```

def is_prime(n):
    if n % 6 not in (1, 5) and n not in (2, 3):
        return False
    for i in range(3, int(math.sqrt(n) + 1)):
        if n % i == 0:
            return False
    return n != 1

```

最终代码如上：

其实是第 2、3 个条件的组合，第 3 个条件可以取代第一个条件。

```

class Solution():
    def isPrime(self, N):
        if N % 6 not in (1, 5) and N not in (2, 3): # 2, 3是例外
            return False
        import math
        for i in range(3, int(math.sqrt(N) + 1)):
            if N != i and N % i == 0:
                return False
        return N != 1 # 1不是素数

```

埃式筛法（素数筛）

就是用筛选出来的素数去过滤所有能够被它整除的数。

这些素数就像是筛子一样去过滤自然数，最后被筛剩下的自然数就是不能被前面素数整除的数，根据素数的定义，这些剩下的数也是素数。

```
class Solution():
    def primeSelect(self, N):
        isPrime = [True for _ in range(N + 1)] # 初始化全为素数，接着进行筛选
        prime = []
        for i in range(2, N + 1): # 从2的倍数开始进行筛选
            if isPrime[i]:
                prime.append(i)
                for j in range(i * 2, N + 1, i): # 步长为 i
                    isPrime[j] = False
        return prime
```

我们来分析一下筛法的复杂度，从代码当中我们可以看到，我们一共有两层循环，最外面一层循环固定是遍历 n 次。

而里面的这一层循环遍历的次数一直在变化，并且它的运算次数和素数的大小相关，看起来似乎不太方便计算。

实际上是可以的，根据[素数分布定理](#)以及一系列复杂的运算（相信我，你们不会感兴趣的），我们是可以得出筛法的复杂度是 $O(N \ln \ln N)$ 。

三、极致优化

筛法的复杂度已经非常近似 $O(n)$ 了，因为即使在 n 很大的时候，经过两次 \ln 的计算，也非常近似常数了，实际上在绝大多数使用场景当中，上面的算法已经足够应用了。

但是仍然有大牛不知满足，继续对算法做出了优化，将其优化到了的复杂度 $O(n)$ 。

虽然从效率上来看并没有数量级的提升，但是应用到的思想非常巧妙，值得我们学习。

在我们理解这个优化之前，先来看看之前的筛法还有什么可以优化的地方。

比较明显地可以看出来，对于一个合数而言，**它可能会被多个素数筛去**。比如 38，它有 2 和 19 这两个素因数，那么它就会被置为两次 `False`，这就带来了**额外的开销**，如果对于每一个合数我们只更新一次，那么是不是就能优化到了呢？

怎么样保证每个合数只被更新一次呢？这里要用到一个定理，就是**每个合数分解质因数只有的结果是唯一的**。

既然是唯一的，那么一定可以找到最小的质因数，如果我们能够保证一个合数只会被它最小的质因数更新为 `False`，那么整个优化就完成了。

其实也不难，我们假设整数 n 的最小质因数是 m ，那么我们用小于 m 的素数 i 乘上 n 可以得到一个合数。

我们将这个合数消除，**对于这个合数而言， i 一定是它最小的质因数**。因为它等于 $i * n$ ， n 最小的质因数是 m ， i 又小于 m ，所以 i 是它最小的质因数，我们用这样的方法来生成消除的合数，这样来保证每个合数只会被它最小的质因数消除。

根据这一点，我们可以写出新的代码：

```
● ● ●

def ertosthenes(n):
    primes = []
    is_prime = [True] * (n+1)
    for i in range(2, n+1):
        if is_prime[i]:
            primes.append(i)
            for j, p in enumerate(primes):
                # 防止越界
                if p > n // i:
                    break
                # 过滤
                is_prime[i * p] = False
                # 当 i % p 等于0的时候说明p就是i最小的质因数
                if i % p == 0:
                    break
    return primes
```

四、总结

到这里，我们关于埃式筛法的介绍就告一段落了。

埃式筛法的优化版本相对来说要难以记忆一些，如果记不住的话，可以就只使用优化之前的版本，两者的效果相差并不大，完全在可以接受的范围之内。

筛法看着代码非常简单，但是**非常重要**，有了它，我们就可以在短时间内获得大量的素数，快速地获得一个素数表。

有了素数表之后，很多问题就简单许多了，比如因数分解的问题，比如信息加密的问题等等。我每次回顾筛法算法的时候都会忍不住感慨，这个两千多年前被发明出来的算法至今看来非但不过时，仍然还是那么巧妙。

希望大家都能怀着崇敬的心情，理解算法当中的精髓。

质数因子

题目描述

功能:输入一个正整数，按照从小到大的顺序输出它的所有质因子（重复的也要列举）（如180的质因子为2 2 3 3 5）
最后一个数后面也要有空格

输入描述:

输入一个long型整数

输出描述:

按照从小到大的顺序输出它的所有质数的因子，以空格隔开。最后一个数后面也要有空格。

示例1

输入	复制
180	
输出	复制
2 2 3 3 5	

思路：毫无疑问，这会是一道经典题，如何求质因子呢？其实很简单，每次都从2开始，去整除180，得到新的数，再从2开始进行整数，因为你是从小开始的，所以你得到的数都会是质数，也就都会是质因子，直到你除出来的数找不到了质因子，说明本身就是质因子了，所以这是一道利用数学思想的题目。

```

class Solution():
    def findZhi(self, n): # 超时
        N = n
        if n == 1 or n == 2:
            print(n, end=" ")
        while n != 1:
            for i in range(2, N):
                if n % i == 0: # 找到一个质因子
                    print(i, end=" ")
                    n = n // i
                    break

    def findZhi(self, n): # 找n的质因子
        flag = True # 设置标志，看是否到了最后一个质因子
        for i in range(2, n):
            if n % i == 0: # 找到一个质因子
                print(i, end=" ")
                self.findZhi(n // i)
                flag = False
                break # 找到了不再找了
        if flag:
            print(n, end=" ") # 这是一个不回溯的过程，用来打印

```

没明白为啥第一种方法会超时，第二种不会。（华为机试）

数值的整数次方

a 是底数,N 是指数,b 是取模数,求 a 的 n 次幂,时间复杂度为 O(log2N)

剑指 Offer 16. 数值的整数次方

难度 中等 39 收藏 分享 切换为英文 关注 反馈

实现函数double Power(double base, int exponent)，求base的exponent次方。不得使用库函数，同时不需要考虑大数问题。

示例 1:

输入: 2.00000, 10
输出: 1024.00000

示例 2:

输入: 2.10000, 3
输出: 9.26100

示例 3:

输入: 2.00000, -2
输出: 0.25000
解释: $2^{-2} = 1/2^2 = 1/4 = 0.25$

```
class Solution:  
    def Power(self, base, exponent):  
        result = 1  
        flag = 1 if exponent > 0 else -1 # 先判断正负  
        exponent = exponent if flag == 1 else -exponent  
        while exponent != 0: # 把指数化成二进制，当指数中没有1的时候，结束循环  
            if exponent & 1 == 1: # 例如: 10^1101 = 10^0001 * 10^0100 * 10^1000。  
                result *= base  
            base = base * base  
            exponent >>= 1 # 等价于 exponent /= 2  
        return result if flag == 1 else 1 / result
```

判断一个数的阶乘的末尾有几个 0

我的思路：能产生 0 的质数组合只有 2×5 ，然后问题就转变成了对 $N!$ 进行质数分解后，一共有几个 5，因为 2 的个数显然多于 5。比如计算 $25!$ 的末尾 0 的个数，包含 5 的数有 5, 10, 15, 20, 25，其中 25 中包含两个 5，所以一共包含 6 个 5。 $25!$ 的末尾有 6 个 0。

```
class Solution():
    def judgeZero(self, n):
        count = 0
        for i in range(5, n + 1):
            while i % 5 == 0: # 判断是否还有5
                count += 1
                i /= 5 # 去除5
        return count
```

直线上最多的点数

149. 直线上最多的点数

难度 困难 153 收藏 分享 切换为英文 关注 反馈

给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。

示例 1：

```
输入: [[1,1],[2,2],[3,3]]
输出: 3
解释:
^
|
|       o
|       o
|   o
+----->
0 1 2 3 4
```

示例 2：

```
输入: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
输出: 4
解释:
^
|
|       o
|       o       o
|       o
|   o       o
+----->
0 1 2 3 4 5 6
```

解题思路：分别以数组中的点作为起始点，其后的点相对于起始点求斜率，斜率相等说明可能

在同一条直线上，计数不同斜率的值的数目，取最大值，然后换一个起始点反复计算。这里需要

注意斜率不能为 0, 所以平行于 x 轴的线段需要另算。最后+1 要算上基准点也在直线上。

```
class Solution:
    def maxPoints(self, points):
        if len(points) == 1:
            return 1
        max_count = 0
        for i in range(len(points)): # 固定第一个点，第二个点在第一个点后，求所有可能的斜率
            xiely = dict() # key 存斜率，val存出现的次数
            infinite = 0
            for j in range(i + 1, len(points)):
                dx = points[i][0] - points[j][0]
                dy = points[i][1] - points[j][1]
                if dx == 0:
                    infinite += 1
                    continue
                g = dy / dx
                xiely[g] = xiely[g] + 1 if g in xiely else 1 # 字典进行初始化
            for key in xiely.keys():
                max_count = max(max_count, xiely[key] + 1) # +1 是算上第一个点本身
            max_count = max(infinite + 1, max_count)
        return max_count
```

只适用于不重复的点，你要是重复了，我也没办法。

三数之和为 N , 三数的最大公约数为 K (阿里)

给定 N 和 K , 求互不相同的正整数 x,y,z 使得 $x+y+z=N$, 且 $\text{gcd}(x,y)=\text{gcd}(x,z)=\text{gcd}(y,z)=K$

思路 : $(x + y + z) * k == N$, 求三个互质的 x、y、z

当 $N // k$ 为偶数 , $x = 1, y = (N // k - 1) // 2, z = y + 1 \rightarrow y, z$ 为一奇一偶

举例 : $N // k = 6, x = 1, y = 2, z = 3$

当 $N // k$ 为奇数 , $x = 1, y + z = N // k - 1$ 偶数 $\rightarrow y, z$ 都为奇数 ,

```
If (N // k - 1) % 2 == 0
    y = (N // k - 1) // 2 - 1, z = (N // k + 1) // 2 + 1
Else:
    y = (N // k - 1) // 2 - 2, z = (N // k + 1) // 2 + 2
```

举例 : $N // k = 13, x = 1, y = 5, z = 7$

$N // k = 15, x = 1, y = 5, z = 9$

最小方案种数（阿里）

题目 1：给定一个数 x ，数据对 (a, b) 使得 $a^b \cdot b^x$ 能达到最大，求使 $|a - b|$ 最小的方案总数有多少。

x, a, b 的范围都是 $0 - (2^{31} - 1)$

思路新颖(模拟)：

Z 字形变换

6. Z 字形变换

难度 中等 777 收藏 分享 切换为英文 关注 反馈

将一个给定字符串根据给定的行数，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 "LEETCODEISHIRING" 行数为 3 时，排列如下：

L	C	I	R				
E	T	O	E	S	I	I	G
E	D	H	N				

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如："LCIRETOESIIGEDHN"。

请你实现这个将字符串进行指定行数变换的函数：

```
string convert(string s, int numRows);
```

示例 1：

```
输入: s = "LEETCODEISHIRING", numRows = 3
输出: "LCIRETOESIIGEDHN"
```

示例 2：

```
输入: s = "LEETCODEISHIRING", numRows = 4
输出: "LDREOEIIECIHNTSG"
解释:
```

L	D	R		
E	O	E	I	I
E	C	I	H	N
T	S	G		

思路：不要被题目复杂的结构所迷惑，其实就是一个从上到下，再从下到上的模拟题

```

class Solution:
    def convert(self, s: str, numRows: int) -> str:
        N, res = len(s), ["" for _ in range(numRows)]
        if numRows <= 1: # 行数为1时，返回自身
            return s
        i, flag = 0, -1
        for x in s:
            res[i] += x
            if i == 0 or i == numRows - 1: flag = -flag # 先确定步长
            i += flag # 再确定下一个位置
        return ''.join(res)

```

加一

66. 加一

难度 简单 | 522 | 收藏 | 分享 | 切换为英文 | 关注 | 反馈

给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位，数组中每个元素只存储单个数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1:

输入: [1, 2, 3]
输出: [1, 2, 4]
解释: 输入数组表示数字 123。

示例 2:

输入: [4, 3, 2, 1]
输出: [4, 3, 2, 2]
解释: 输入数组表示数字 4321。

```

class Solution:
    def plusOne(self, digits: List[int]) -> List[int]:
        carry, res, count, N = 0, [], 0, len(digits)
        while carry or count == 0: # count == 0是为了能够进入循环
            d = digits.pop() if digits else 0
            if count == 0: # 只需要个位数加1即可
                cur = (d + 1 + carry) % 10
                carry = (d + 1 + carry) // 10
            else:
                cur = (d + carry) % 10
                carry = (d + carry) // 10
            res = [cur] + res
            count += 1
        return digits[:N - count] + res

```

灯泡开关 IV

1529. 灯泡开关 IV

难度 中等 例 9 收藏 分享 切换为英文 关注 反馈

房间里有 n 个灯泡，编号从 0 到 $n-1$ ，自左向右排成一行。最开始的时候，所有的灯泡都是关着的。

请你设法使得灯泡的开关状态和 target 描述的状态一致，其中 $target[i]$ 等于 1 第 i 个灯泡是开着的，等于 0 意味着第 i 个灯是关着的。

有一个开关可以用于翻转灯泡的状态，翻转操作定义如下：

- 选择当前配置下的任意一个灯泡（下标为 i ）
- 翻转下标从 i 到 $n-1$ 的每个灯泡

翻转时，如果灯泡的状态为 0 就变为 1，为 1 就变为 0。

返回达成 target 描述的状态所需的最少翻转次数。

示例 1：

```
输入：target = "10111"
输出：3
解释：初始配置 "00000".
从第 3 个灯泡（下标为 2）开始翻转 "00000" -> "00111"
从第 1 个灯泡（下标为 0）开始翻转 "00111" -> "11000"
从第 2 个灯泡（下标为 1）开始翻转 "11000" -> "10111"
至少需要翻转 3 次才能达成 target 描述的状态
```

思路：如果当前位置与前一个位置不同，表明当前位置比前一个位置多经历了一次翻转。

```
class Solution:
    def minFlips(self, target: str) -> int:
        N, pre, count = len(target), "0", 0 # pre定义的巧妙
        for i in range(N):
            if target[i] != pre:
                count += 1
            pre = target[i]
        return count
```

灯泡开关 III

1375. 灯泡开关 III

难度 中等 23 收藏 分享 切换为英文 关注 反馈

房间里有 n 枚灯泡，编号从 1 到 n ，自左向右排成一排。最初，所有的灯都是关着的。

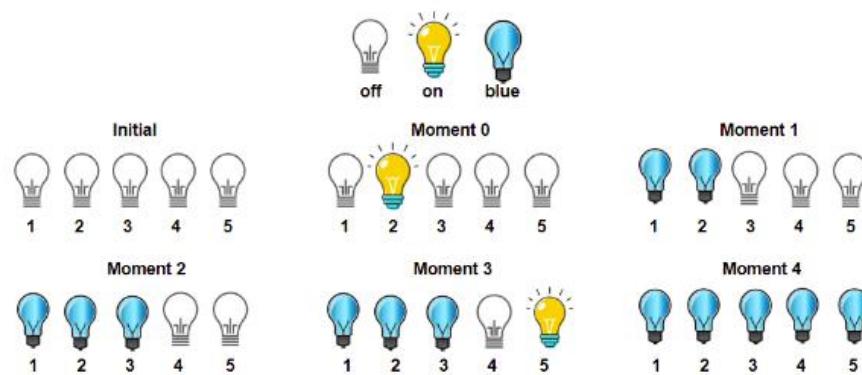
在 k 时刻（ k 的取值范围是 0 到 $n - 1$ ），我们打开 $\text{light}[k]$ 这个灯。

灯的颜色要想 变成蓝色 就必须同时满足下面两个条件：

- 灯处于打开状态。
- 排在它之前（左侧）的所有灯也都处于打开状态。

请返回能够让 所有开着的 灯都 变成蓝色 的时刻 数目。

示例 1：



输入：light = [2, 1, 3, 5, 4]

输出：3

解释：所有开着的灯都变蓝的时刻分别是 1, 2 和 4。

解题思路：你遍历数组，不论如何，你都是要亮一个灯的，这个时候你沿途记录亮的灯中最

右边的那个灯，也就是最大灯索引，当遍历的灯数 == 最大灯索引，那么就说明左边的灯

全亮了。

```
class Solution:  
    def numTimesAllBlue(self, light: List[int]) -> int:  
        N, count, curMax = len(light), 0, 0  
        for i in range(N):  
            curMax = max(curMax, light[i])  
            if curMax == i + 1: # 遍历过的灯数等于索引加1  
                count += 1  
        return count
```

三个数的最大乘积

628. 三个数的最大乘积

难度 简单 150 收藏 分享 切换为英文 关注 反馈

给定一个整型数组，在数组中找出由三个数组成的最大乘积，并输出这个乘积。

示例 1:

输入: [1, 2, 3]
输出: 6

示例 2:

输入: [1, 2, 3, 4]
输出: 24

注意:

1. 给定的整型数组长度范围是 $[3, 10^4]$ ，数组中所有的元素范围是 $[-1000, 1000]$ 。
2. 输入的数组中任意三个数的乘积不会超出32位有符号整数的范围。

我们将数组进行升序排序，如果数组中所有的元素都是非负数，那么答案即为最后三个元素的乘积。

如果数组中出现了负数，那么我们还需要考虑乘积中包含负数的情况，显然选择最小的两个负数和最大的一个正数是最优的，即为前两个元素与最后一个元素的乘积。

上述两个结果中的较大值就是答案。注意我们可以不用判断数组中到底有没有正数，0 或者负数，因为上述两个结果实际上已经包含了所有情况，最大值一定在其中。

```
class Solution:  
    def maximumProduct(self, nums: [int]) -> int:  
        nums.sort()  
        return max(nums[-1] * nums[-2] * nums[-3], nums[0] * nums[1] * nums[-1])
```

最大数

179. 最大数

难度 中等 335 收藏 分享 切换为英文 关注 反馈

给定一组非负整数，重新排列它们的顺序使之组成一个最大的整数。

示例 1：

```
输入: [10, 2]
输出: 210
```

示例 2：

```
输入: [3, 30, 34, 5, 9]
输出: 9534330
```

说明：输出结果可能非常大，所以你需要返回一个字符串而不是整数。

```
class LargerNumKey(str):
    def __lt__(x, y):
        return x + y > y + x # 两个字符串的大小取决于看谁在前面字符串更大

class Solution:
    def largestNumber(self, nums: [int]) -> str:
        strNums = list(map(str, nums))
        strNums = sorted(strNums, key=LargerNumKey)
        return "" .join(strNums) if strNums[0] != '0' else '0'
```

最大余子数组（阿里）

给定一个 n, m ，其中 n 为数组中元素的个数， m 为限定值，再给了一个数组，从数组中挑选一个子序列，和对 m 取余最大，求最大余数。

```

n,m=map(int,input().split())
a=list(map(int,input().split()))
temlist=[0 for i in range(n+1)]
temlist[0]=0
for i in range(n):
    temmax=max(temlist[i],(temlist[i]+a[i])%m)
    for j in temlist[:i]:
        if i+1!=n and temmax>j+a[i+1]:
            continue
        elif i+1!=n and temmax< j+a[i+1] and j+a[i+1]<m:
            temmax=j+a[i+1]
        else:
            break
    temlist[i+1]=temmax
#print(temlist)
print(temlist[-1])

```

幸运数（阿里）

求区间 $[l, r]$ 内的幸运数。幸运数定义为，将相邻数位差的绝对值拼成下一个数，重复该操作直到只剩 1 位。剩下 7 的是幸运数。例如， $219 \rightarrow 18 \rightarrow 7$ 或者 $118 \rightarrow 7$

条件： $1 \leq l \leq r \leq 1e9$

思路：根据 $1, \dots, k-1$ 位的幸运数，给定首位数字，可以搜索得到 k 位的幸运数。将所有幸运数排序后进行二分查找。

因为每一个更长的幸运数都可以通过题目里规定的操作变成一个更短的幸运数，反过来讲，给定每个幸运数的首位 `for i in {1, ..., 9}`，它都可以通过一个比他短的幸运数计算得到。

比如给定首位是 1，可以算 $7 \rightarrow 18$ 。根据 18，给定首位是 2，又可以算出 $18 \rightarrow 219$ 。

不断这样以小算大，可以算出所有幸运数。

注意 k 位的幸运数不仅可以通过 $k-1$ 的幸运数计算得到，还可以通过更短的幸运数在前方补 0 到 $k-1$ 位后计算得到。例如：7 补成 007，可以计算得到 1118, 2229, 7770, 8881, 9992。

也就是说需要你用造数的思想，而不是遍历数的思想去做题。

扑克牌中的顺子

剑指 Offer 61. 扑克牌中的顺子

难度 简单 40 收藏 分享 切换为英文 关注 反馈

从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2~10为数字本身，A为1，J为11，Q为12，K为13，而大、小王为0，可以看成任意数字。A不能视为14。

示例 1：

输入：[1, 2, 3, 4, 5]
输出：True

示例 2：

输入：[0, 0, 1, 2, 5]
输出：True

理解错题意，就是问你抽五张牌，大小王可替代，能否凑成顺子！

```
class Solution:
    def isStraight(self, nums: [int]) -> bool:
        Max, Min = 0, 14
        already = []
        for x in nums:
            if x == 0:
                continue
            if x in already: # 除大王外不能重复
                return False
            already.append(x)
            Max = max(Max, x)
            Min = min(Min, x)
        return Max - Min < 5 # 五张牌中最大值和最小值之差小于5
```

咖啡的香气（大疆）

咖啡的香气

时间限制：C/C++语言 1000MS；其他语言 3000MS

内存限制：C/C++语言 65536KB；其他语言 589824KB

题目描述：

自从零食间开始免费供应上好的咖啡豆，小杰每天午休后都会来到零食间，按下咖啡机的按钮，等待着杯里弥漫开来的香气把自己淹没，纷乱的思绪也渐渐在水雾中模糊。“小杰，你还有N个bug没修，别摸鱼了，快来解bug！”，一个不合时宜的声音往往会在此时响起，小杰的脑海中瞬间闪过了无数个文件，无数行代码随着咖啡的香气不断滚动。

“我是不可能写bug的，这辈子都不可能写bug的...”，小杰一边念叨着，一边开始在脑海里盘算起来。

假设每喝一杯咖啡（喝咖啡的时间忽略不计），就能让自己一小时内的debug效率提升到原来的A倍，一小时内重复喝没用，最多只能喝X杯，太多了晚上会睡不着，并且为了保证可持续发展，每天最多只能专注工作8个小时，而在没喝咖啡的状态下解决每个bug所需要的时间为 $t_1, t_2 \dots t_N$ 分钟。

小杰的咖啡还没有喝完，你能帮他计算出他今天能解完所有bug吗？如果能，最少需要多长时间？

思路：不能总想着把一小时拆开，而是应该翻倍，也就是说花一个小时能做 effect 小时的事。那么首先统计总共需要多久时间完成任务，在咖啡的高效时间内，减去了多少时间，然后再算剩余的普通时间是否大于完成任务所需的时间，存在三种情况：

- 1、在高效时间内就完成了所有任务
- 2、在高效时间内未完成所有任务，普通时间能完成任务。
- 3、在高效时间内未完成所有任务，普通时间不能完成任务。

```

class Solutions():
    def finish(self, effect, coffeeNum, bugTime):
        if coffeeNum > 8:
            coffeeNum = 8
        totalTime, totalSpend = 0, 0
        for x in bugTime:
            totalTime += x
        while coffeeNum != 0 and totalTime > 0:
            totalTime -= 60 * effect      # 工作一小时相当于减少effect小时的工作量
            coffeeNum -= 1
            totalSpend += 60
        if coffeeNum == 0 and totalTime - (8 * 60 - totalSpend) > 0:
            return 0
        elif totalTime <= 0:# 说明提前完成了
            return totalSpend + totalTime // effect
        else:
            return totalSpend + totalTime

if __name__ == '__main__':
    bugNum, effect, coffeeNum = list(map(int, input().split()))
    bugTime = list(map(int, input().split()))
    solution = Solutions()
    print(solution.finish(effect, coffeeNum, bugTime))

```

根据身高重建队列

406. 根据身高重建队列

难度 中等 371 收藏 分享 切换为英文 关注 反馈

假设有打乱顺序的一群人站成一个队列。每个人由一个整数对 (h, k) 表示，其中 h 是这个人的身高， k 是排在这个人前面且身高大于或等于 h 的人数。编写一个算法来重建这个队列。

注意：

总人数少于1100人。

示例

输入：
`[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]`

输出：
`[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]`

通过次数 32,392 | 提交次数 49,789

很明显这道题目并不是真的按照身高进行排序 ,而是根据身高和前面比它的身高高的人进行排序 ,所以你不能先将矮个子排序 ,因为题目中的第二个参数是 ,前面比它高的人的人

数，如果你从最高的开始排序，那么你就能够完美知道哪些人比你高，你就可以站到指定的位置。

换言之，这是一道你只知道比你高的人有多少个的题目。

本质上就是插入排序的变种，因为这个插入排序利用了新的空间，而不是就地排序。

```
class Solution:
    def reconstructQueue(self, people: [[int]]) -> [[int]]:
        people = sorted(people, key=lambda x : (-x[0], x[1])) # 按身高逆序，按人数升序
        N = len(people)
        res = []
        for i in range(N):
            res.insert(people[i][1], people[i]) # insert 某个位置时 res可以为空
        return res
```

有效的数独

36. 有效的数独

难度 中等 击 349 收藏 分享 切换为英文 关注 反馈

判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。

1. 数字 1-9 在每一行只能出现一次。
2. 数字 1-9 在每一列只能出现一次。
3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

5	3	.	.	7
6	.	.	1	9	5	.	.	.
.	9	8	.	.	.	6	.	.
8	.	.	6	.	.	.	3	.
4	.	8	.	3	.	.	.	1
7	.	.	2	6
.	6	.	.	.	2	8	.	.
.	.	4	1	9	.	.	.	5
.	.	.	8	.	7	9	.	.

上图是一个部分填充的有效的数独。

数独部分空格内已填入了数字，空白格用 ‘.’ 表示。

示例 1：

输入：

```
[  
    ["5", "3", ".", ".", "7", ".", ".", ".", "."],  
    ["6", ".", ".", "1", "9", "5", ".", ".", "."],  
    [".", "9", "8", ".", ".", ".", "6", "."],  
    ["8", ".", ".", "6", ".", ".", ".", "3"],  
    ["4", ".", ".", "8", ".", "3", ".", ".", "1"],  
    ["7", ".", ".", "2", ".", ".", ".", "6"],  
    [".", "6", ".", ".", ".", "2", "8", "."],  
    [".", ".", "4", "1", "9", ".", ".", "5"],  
    [".", ".", "8", ".", ".", "7", "9"]  
]
```

输出：true

示例 2:

输入:

```
[["8", "3", ".", ".", "7", ".", ".", ".", "."],  
 ["6", ".", ".", "1", "9", "5", ".", ".", "."],  
 [".", "9", "8", ".", ".", ".", "6", "."],  
 ["8", ".", ".", ".", "6", ".", ".", ".", "3"],  
 [".", ".", "8", ".", "3", ".", ".", "1"],  
 [".", ".", ".", "2", ".", ".", "6"],  
 [".", "6", ".", ".", "2", "8", "."],  
 [".", ".", "4", "1", "9", ".", ".", "5"],  
 [".", ".", "8", ".", "7", "9"]]
```

]

输出: false

解释: 除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与 示例1 相同。

但由于位于左上角的 3x3 宫内有两个 8 存在，因此这个数独是无效的。

说明:

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。
- 给定数独序列只包含数字 1-9 和字符 ‘.’。
- 给定数独永远是 9x9 形式的。

对每一行，每一列，每一个盒子都建立一张 hash 表，也就是 27 张 hash 表，存入三个数组 row、col、box，row[i]表示第一行的元素的 hash 表，col[j]表示第一列的元素的 hash 表，box[(i // 3) * 3 + (j // 3)]表示第几个盒子的 hash 表。

建立一个二层循环，依次存入 hash 表，hash 表中存在即 return False。

```
class Solution:  
    def isValidSudoku(self, board: [[str]]) -> bool:  
        # 27张hash表，9张存行，9张存列，9张存9宫格  
        row, col, box = [[] for _ in range(9)], [[] for _ in range(9)], [[] for _ in range(9)]  
        print(row)  
        for i in range(9):  
            for j in range(9):  
                if board[i][j] != ".":  
                    if board[i][j] in row[i]:  
                        return False  
                    elif board[i][j] in col[j]:  
                        return False  
                    elif board[i][j] in box[(i // 3) * 3 + (j // 3)]:  
                        return False  
                    row[i].append(board[i][j])  
                    col[j].append(board[i][j])  
                    box[(i // 3) * 3 + (j // 3)].append(board[i][j]) # 行号 * 列数 + 列号  
        return True
```

需要的最小会议室的数目

```
class Interval(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end
class Solution:
    def minMeetingRooms(self, intervals):
        size = len(intervals)
        point = [] # 存取所有时刻，开始时刻和结束时刻
        for interval in intervals:
            point.append((interval.start, 1)) # 1表示会议室开启
            point.append((interval.end, -1)) # -1表示会议室关闭
        on_MeetingRoom = 0
        max_MeetingRoom = 0
        for _, delta in sorted(point): # 将所有时间点排好序
            on_MeetingRoom += delta
            max_MeetingRoom = max(on_MeetingRoom, max_MeetingRoom)
        return max_MeetingRoom

#主函数
if __name__ == '__main__':
    node1 = Interval(0, 30)
    node2 = Interval(5, 10)
    node3 = Interval(15, 20)
    intervals = [node1, node2, node3]
    solution = Solution()
    print(solution.minMeetingRooms(intervals))
```

```
"D:\Program Files\Python37\venv\Scripts\python.exe" F:/algos/algorithm/meetingRooms.py
[(0, 1), (5, 1), (10, -1), (15, 1), (20, -1), (30, -1)]
2
```

回文排列

面试题 01.04. 回文排列

难度 简单 25 收藏 分享 切换为英文 关注

给定一个字符串，编写一个函数判定其是否为某个回文串的排列之一。

回文串是指正反两个方向都一样的单词或短语。排列是指字母的重新排列。

回文串不一定是字典当中的单词。

示例1：

输入："tactcoa"
输出：true (排列有"tacocat"、"atcocta"，等等)

计数字母，如果字母的个数存在两个奇数，表明无法组成回文排列，存在一个奇数，可以放在中间组成回文排列。

```
class Solution:
    def canPermutePalindrome(self, s):
        lookup = dict()
        count = 0
        for x in s:# 遍历字符串
            if x not in lookup:
                lookup[x] = 1# 初始化
            else:
                lookup[x] += 1
        for x in lookup.values():
            count += x % 2# 计数奇数的个数
        return count <= 1 # 要么只有一个奇数，要么没有奇数

class Solution:
    def canPermutePalindrome(self, s):
        count = 0
        lookup = collections.Counter(s) # 计数字符串中字符的个数。返回以字符为key的字典
        print(lookup)
        for x in lookup.values():
            count += x % 2# 计数奇数的个数
        return count <= 1 # 要么只有一个奇数，要么没有奇数
```

```
输入的字符串是：s= abbv  
Counter({'b': 2, 'v': 1, 'a': 1})  
输出的结果是： False
```

Collections.Counter()也可以计数列表

```
List = [1, 2, 1]  
look = collections.Counter(List)  
print(look)
```

```
Counter({1: 2, 2: 1})
```

分糖果

575. 分糖果

难度 简单 74 收藏 分享 切换为英文 关注 反馈

给定一个偶数长度的数组，其中不同的数字代表着不同种类的糖果，每一个数字代表一个糖果。你需要把这些糖果平均分给一个弟弟和一个妹妹。返回妹妹可以获得的最大糖果的种类数。

示例 1：

```
输入: candies = [1,1,2,2,3,3]  
输出: 3  
解析: 一共有三种种类的糖果，每一种都有两个。  
最优分配方案：妹妹获得[1, 2, 3]，弟弟也获得[1, 2, 3]。这样使妹妹获得糖果的种类数最多。
```

示例 2：

```
输入: candies = [1,1,2,3]  
输出: 2  
解析: 妹妹获得糖果[2, 3]，弟弟获得糖果[1, 1]，妹妹有两种不同的糖果，弟弟只有一种。这样使得妹妹可以获得的糖果种类数最多。
```

注意：

1. 数组的长度为[2, 10,000]，并且确定为偶数。
2. 数组中数字的大小在范围[-100,000, 100,000]内。

```
class Solution:  
    def distributeCandies(self, candies: [int]) -> int:  
        return len(set(candies)) if len(set(candies)) <= len(candies)//2 else len(candies) // 2  
        # 种类数小于糖果的一半，表明有些糖果不止一个，那么妹妹每种糖果都可以有一个，所以是len(set(candies))  
        # 种类数大于糖果的一半，说明有些糖果只有一个，那么妹妹可以得到len(candies)//2的糖果数，其余糖果均无法得到。
```

分糖果 2

135. 分发糖果

难度 困难 223 收藏 分享 切换为英文 关注 反馈

老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。

你需要按照以下要求，帮助老师给这些孩子分发糖果：

- 每个孩子至少分配到 1 个糖果。
- 相邻的孩子中，评分高的孩子必须获得更多的糖果。

那么这样下来，老师至少需要准备多少颗糖果呢？

示例 1：

输入： [1, 0, 2]
输出： 5
解释： 你可以分别给这三个孩子分发 2、1、2 颗糖果。

示例 2：

输入： [1, 2, 2]
输出： 4
解释： 你可以分别给这三个孩子分发 1、2、1 颗糖果。
第三个孩子只得到 1 颗糖果，这已满足上述两个条件。

```
class Solution:  
    def candy(self, ratings):  
        candynum = [1 for i in range(len(ratings))] # 每个小孩至少有一个糖果  
        for i in range(1, len(ratings)):  
            if ratings[i] > ratings[i - 1]:  
                candynum[i] = candynum[i - 1] + 1 # 如果右边的排名更高，在前者的基础上加一个糖果  
        for i in range(len(ratings) - 2, -1, -1):  
            if ratings[i + 1] < ratings[i] and candynum[i + 1] >= candynum[i]:  
                candynum[i] = candynum[i + 1] + 1 # 如果左边的排名更高，在后者的基础上加一个糖果  
        return sum(candynum)
```

下一个更大的元素 I

496. 下一个更大元素 I

难度 简单 232 收藏 分享 切换为英文 关注 反馈

给定两个 **没有重复元素** 的数组 `nums1` 和 `nums2`，其中 `nums1` 是 `nums2` 的子集。找到 `nums1` 中每个元素在 `nums2` 中的下一个比其大的值。

`nums1` 中数字 `x` 的下一个更大元素是指 `x` 在 `nums2` 中对应位置的右边的第一个比 `x` 大的元素。如果不存在，对应位置输出 `-1`。

示例 1:

输入: `nums1 = [4,1,2]`, `nums2 = [1,3,4,2]`.

输出: `[-1,3,-1]`

解释:

对于 `num1` 中的数字 4，你无法在第二个数组中找到下一个更大的数字，因此输出 `-1`。

对于 `num1` 中的数字 1，第二个数组中数字 1 右边的下一个较大数字是 3。

对于 `num1` 中的数字 2，第二个数组中没有下一个更大的数字，因此输出 `-1`。

示例 2:

输入: `nums1 = [2,4]`, `nums2 = [1,2,3,4]`.

输出: `[3,-1]`

解释:

对于 `num1` 中的数字 2，第二个数组中的下一个较大数字是 3。

对于 `num1` 中的数字 4，第二个数组中没有下一个更大的数字，因此输出 `-1`。

```
class Solution:
    def nextGreaterElement(self, nums1: [int], nums2: [int]) -> [int]:
        M, N = len(nums1), len(nums2)
        result = [0] * M
        for i in range(M):
            flag = False
            index = nums2.index(nums1[i]) # 不是对应位置，而是对应元素
            for j in range(index + 1, N):
                if nums2[j] > nums1[i]:
                    result[i] = nums2[j]
                    flag = True
                    break
            if not flag:
                result[i] = -1
        return result
```

下一个更大的元素 2

503. 下一个更大元素 II

难度 中等 158 收藏 分享 切换为英文 关注 反馈

给定一个循环数组（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 -1。

示例 1：

```
输入: [1,2,1]
输出: [2,-1,2]
解释: 第一个 1 的下一个更大的数是 2;
数字 2 找不到下一个更大的数;
第二个 1 的下一个最大的数需要循环搜索，结果也是 2。
```

注意：输入数组的长度不会超过 10000。

```
class Solution:
    def nextGreaterElements(self, nums: [int]) -> [int]:
        N = len(nums)
        result = [0] * N
        for i in range(N):
            flag = False
            for j in range(1, N):
                if nums[(i + j) % N] > nums[i]: # 采用取余的方式即可
                    result[i] = nums[(i + j) % N]
                    flag = True
                    break
            if not flag:
                result[i] = -1
        return result
```

加油站

134. 加油站

难度 中等 324 收藏 分享 切换为英文 关注 反馈

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 $\text{gas}[i]$ 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 $i+1$ 个加油站需要消耗汽油 $\text{cost}[i]$ 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

- 如果题目有解，该答案即为唯一答案。
- 输入数组均为非空数组，且长度相同。
- 输入数组中的元素均为非负数。

示例 1:

输入:

```
gas = [1, 2, 3, 4, 5]
cost = [3, 4, 5, 1, 2]
```

输出: 3

解释:

从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 $= 0 + 4 = 4$ 升汽油

开往 4 号加油站，此时油箱有 $4 - 1 + 5 = 8$ 升汽油

开往 0 号加油站，此时油箱有 $8 - 2 + 1 = 7$ 升汽油

开往 1 号加油站，此时油箱有 $7 - 3 + 2 = 6$ 升汽油

开往 2 号加油站，此时油箱有 $6 - 4 + 3 = 5$ 升汽油

开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。

因此，3 可为起始索引。

```
class Solution:
    def canCompleteCircuit(self, gas: [int], cost: [int]) -> int:
        N = len(gas)
        for i in range(N):# 起始点
            gasSum = 0
            for j in range(N):
                pos = (i + j) % N # 通过
                gasSum += gas[pos] - cost[pos]
                if gasSum < 0:
                    break
                if gasSum >= 0:
                    return i
        return -1
```

将每个元素替换成右侧最大元素

1299. 将每个元素替换成右侧最大元素

难度 简单 33 喜欢 16 收藏 10

给你一个数组 `arr`，请你将每个元素用它右边最大的元素替换，如果是最后一个元素，用 `-1` 替换。

完成所有替换操作后，请你返回这个数组。

示例：

```
输入: arr = [17,18,5,4,6,1]
输出: [18,6,6,6,1,-1]
```

如果从左往右的话，因为无法利用到左侧已经拍好序的函数，所以会超时，故应该逆序便利

数组。

```
class Solution:
    def replaceElements(self, arr: [int]) -> [int]:
        N = len(arr)
        result = [0] * N
        result[-1] = -1
        for i in range(N - 2, -1, -1):
            result[i] = max(result[i + 1], arr[i + 1])
        return result
```

矩阵图形

岛屿的最大面积

695. 岛屿的最大面积

难度 中等 316 收藏 分享 切换为英文 关注 反馈

给定一个包含了一些 0 和 1 的非空二维数组 grid。

一个 **岛屿** 是由一些相邻的 1 (代表土地) 构成的组合，这里的「相邻」要求两个 1 必须在水平或者竖直方向上相邻。你可以假设 grid 的四个边缘都被 0 (代表水) 包围着。

找到给定的二维数组中最大的岛屿面积。(如果没有岛屿，则返回面积为 0。)

示例 1:

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],  
 [0,0,0,0,0,0,1,1,1,0,0,0],  
 [0,1,1,0,1,0,0,0,0,0,0,0],  
 [0,1,0,0,1,1,0,0,1,0,1,0],  
 [0,1,0,0,1,1,0,0,1,1,0,0],  
 [0,0,0,0,0,0,0,0,0,1,0,0],  
 [0,0,0,0,0,0,1,1,1,0,0,0],  
 [0,0,0,0,0,0,1,1,0,0,0,0]]
```

对于上面这个给定矩阵应返回 6。注意答案不应该是 11，因为岛屿只能包含水平或垂直的四个方向的 1。

示例 2:

```
[[0,0,0,0,0,0,0]]
```

对于上面这个给定的矩阵，返回 0。

注意: 给定的矩阵 grid 的长度和宽度都不超过 50。

```

class Solution:
    def maxAreaOfIsland(self, grid: [[int]]) -> int:
        dx, dy = [0, 1, 0, -1], [1, 0, -1, 0]
        import collections
        queue = collections.deque()
        M, N, maxArea = len(grid), len(grid[0]), 0
        for i in range(M):
            for j in range(N):
                if grid[i][j] == 1:
                    area = 0
                    queue.append([i, j]) # 先要进队列
                    while queue:
                        x, y = queue.popleft() # 循环后出队列
                        grid[x][y] = 0 # 置0
                        area += 1 # 真正访问的位置
                        for k in range(4):
                            nextX, nextY = x + dx[k], y + dy[k]
                            if not (0 <= nextX < M and 0 <= nextY < N) or grid[nextX][nextY] == 0:
                                continue
                            if (nextX, nextY) not in queue: # 不能存在queue中
                                queue.append((nextX, nextY))
                    maxArea = max(maxArea, area)
        return maxArea

```

螺旋矩阵

54. 螺旋矩阵

难度 中等 399 收藏 分享 切换为英文 关注 反馈

给定一个包含 $m \times n$ 个元素的矩阵 (m 行, n 列)，请按照顺时针螺旋顺序，返回矩阵中的所有元素。

示例 1：

输入：
`[
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
]`
输出： [1, 2, 3, 6, 9, 8, 7, 4, 5]

示例 2：

输入：
`[
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]`
输出： [1, 2, 3, 4, 8, 12, 11, 10, 9, 5, 6, 7]

这种矩阵走路问题，三个必然用到的条件：

1、inbound 函数

2、 $Dx = [0, 1, -1, 0]$, $dy = [1, 0, -1, 0]$

3、Visit 矩阵

方向变化：

1、方向向右，到头，向下转

2、方向向下，到头，向左转

3、方向向左，到头，向上转

4、方向向上，到头，向右转

注意：

当没有用到 queue 时

```
res.append(matrix[nextX][nextY])
visit[nextX][nextY] = 1
```

这两者是同在的。

```

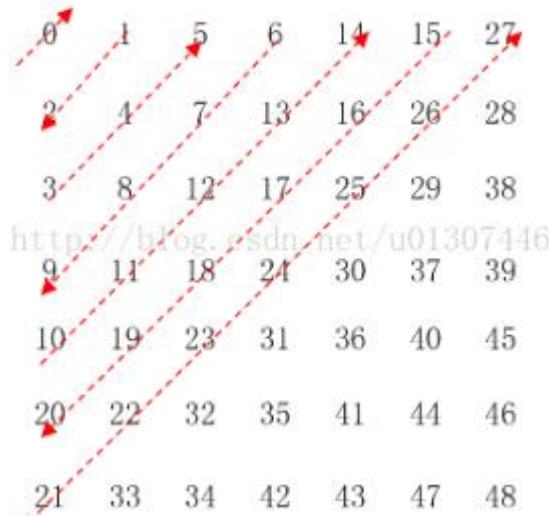
class Solution:
    def spiralOrder(self, matrix: [[int]]) -> [int]:
        dx = [0, 1, 0, -1]
        dy = [1, 0, -1, 0]
        if len(matrix) == 0:
            return []
        M, N = len(matrix), len(matrix[0])
        visit = [[0] * N for _ in range(M)]
        res = []
        x, y = 0, 0
        res.append(matrix[0][0])
        visit[0][0] = 1
        direct = 0
        while len(res) < M * N:
            nextX, nextY = x + dx[direct], y + dy[direct]
            if not self.inbound(matrix, nextX, nextY) or visit[nextX][nextY] == 1:
                if direct == 0:
                    x += 1
                elif direct == 1:
                    y -= 1
                elif direct == 2:
                    x -= 1
                else:
                    y += 1
                direct = (direct + 1) % 4
            else:
                x, y = nextX, nextY
            res.append(matrix[x][y]) # 注意x, y才是主角
            visit[x][y] = 1
        return res

    def inbound(self, matrix, x, y):
        return 0 <= x < len(matrix) and 0 <= y < len(matrix[0])

```

矩阵元素 zigzag 返回

给定一个 m 行、n 列的矩阵，以 ZigZag 的顺序返回矩阵中所有元素，如图所示：



其实相比较螺旋矩阵问题，这道题的话就是只有两个方向：

右上和左下 $dx[-1, 1], dy[1, -1]$

只要分好情况讨论即可：

出界：

方向左下：

下出界：向右；

左出界：向下；

方向改为右上

方向右上：

右出界：向下；

上出界：向右；

方向改为左下

```
class Solution:
    def printZMatrix(self, matrix):
        dx = [-1, 1]
        dy = [1, -1]
        direct = 0
        rowNum, colNum = len(matrix), len(matrix[0])
        result = [matrix[0][0]]
        x, y = 0, 0
        for i in range(rowNum * colNum - 1):
            nextX, nextY = x + dx[direct], y + dy[direct]
            if 0 <= nextX < rowNum and 0 <= nextY < colNum:
                x, y = nextX, nextY
            else:
                if direct == 0:
                    if nextX < 0 and nextY < colNum:
                        y = y + 1 # 只x越界，向右运动
                    else:
                        x = x + 1 # xy都越界或y越界，向下运动
                    direct = 1
                else:
                    if nextY < 0 and nextX < rowNum:
                        x = x + 1 # 只y越界，向下运动
                    else:
                        y = y + 1 # xy都越界或x越界，向右运动
                    direct = 0
            result.append(matrix[x][y]) # 确定好下一个位置的坐标后，存入结果数组
        return result
```

岛屿的周长

463. 岛屿的周长

难度 简单 199 收藏 分享 切换为英文 关注 反馈

给定一个包含 0 和 1 的二维网格地图，其中 1 表示陆地 0 表示水域。

网格中的格子水平和垂直方向相连（对角线方向不相连）。整个网格被水完全包围，但其中恰好有一个岛屿（或者说，一个或多个表示陆地的格子相连组成的岛屿）。

岛屿中没有“湖”（“湖”指水域在岛屿内部且不和岛屿周围的水相连）。格子是边长为 1 的正方形。网格为长方形，且宽度和高度均不超过 100。计算这个岛屿的周长。

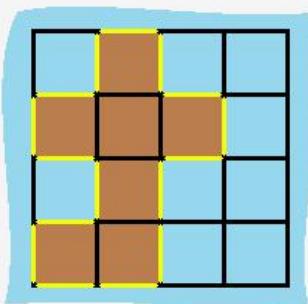
示例：

输入：

```
[[0,1,0,0],  
 [1,1,1,0],  
 [0,1,0,0],  
 [1,1,0,0]]
```

输出：16

解释：它的周长是下面图片中的 16 个黄色的边：



这道题肯定可以用遍历，遇到 1 周长先加 4，再判断其左上有没有 1，有的话自动减 2，

但这毫无疑问当湖足够大，岛屿足够小时，就会浪费很多的时间复杂度。故我决定采用

queue，先找到岛屿的第一个边缘，然后将整个岛屿加入 queue，从 queue 中提取一个

格子时周长加 4，当找到一个新格子时，周长减 2 即可。

```

class Solution:
    def islandPerimeter(self, grid: [[int]]) -> int:
        dx = [0, 1, 0, -1]
        dy = [1, 0, -1, 0]
        if len(grid) == 0:
            return 0
        M, N = len(grid), len(grid[0])
        visit = [[0] * N for _ in range(M)]
        Perim = 0
        import collections
        queue = collections.deque()
        flag = 0 # 用来找入口
        for i in range(M):
            for j in range(N):
                if grid[i][j] == 1:
                    queue.append((i, j))
                    flag = 1
                    break
            if flag == 1:
                break
        while queue:
            x, y = queue.popleft()
            visit[x][y] = 2
            Perim += 4 # 弹出表明开始访问
            for i in range(4):
                nextX, nextY = x + dx[i], y + dy[i]
                if not (0 <= nextX < M and 0 <= nextY < N) or visit[nextX][nextY] == 2 or grid[nextX][nextY] == 0:
                    continue
                if (nextX, nextY) not in queue: # 特别注意，不要将queue中原本存在的再加一遍。
                    queue.append((nextX, nextY))
            Perim -= 2 # 入栈表示相邻
        return Perim

```

被围绕的区域

130. 被围绕的区域

难度 中等 241 收藏 分享 切换为英文 关注 反馈

给定一个二维的矩阵，包含 'X' 和 'O' (字母 O)。

找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例:

```

X X X X
X O O X
X X O X
X O X X

```

运行你的函数后，矩阵变为：

```

X X X X
X X X X
X X X X
X O X X

```

解释:

被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

无法直接找到被围绕的区域有哪些，那么就找没被围绕的区域有哪些，然后将没被围绕的区域的 O 改为 T，然后再将所以非 T 区域改为 X，然后将 T 区域改为 O，就实现了等价

操作。

这里采用 queue 求解，特别适合不规则回溯方法。

```
class Solution:
    def solve(self, board: [[str]]) -> None:
        dx = [0, 1, 0, -1]
        dy = [1, 0, -1, 0]
        if len(board) == 0:
            return []
        M, N = len(board), len(board[0])
        import collections
        queue = collections.deque()
        for i in range(M):
            if board[i][0] == '0':
                queue.append((i, 0))
            if board[i][N - 1] == '0':
                queue.append((i, N - 1))
        for j in range(N):
            if board[0][j] == '0':
                queue.append((0, j))
            if board[M - 1][j] == '0':
                queue.append((M - 1, j)) # 初始化queue
        while queue:
            x, y = queue.popleft()
            board[x][y] = 'T'
            for i in range(4):
                nextX, nextY = x + dx[i], y + dy[i]
                if not (0 <= nextX < M and 0 <= nextY < N) or board[nextX][nextY] != '0':
                    continue
                if (nextX, nextY) not in queue: # 虽然不加对这道题没影响，但是可以降低时间复杂度
                    queue.append((nextX, nextY))
        for i in range(M):
            for j in range(N):
                if board[i][j] == '0':
                    board[i][j] = 'X'
                if board[i][j] == 'T':
                    board[i][j] = '0'
    return board
```

太平洋和大西洋水流问题

417. 太平洋大西洋水流问题

难度 中等 112 收藏 分享 切换为英文 关注 反馈

给定一个 $m \times n$ 的非负整数矩阵来表示一片大陆上各个单元格的高度。“太平洋”处于大陆的左边界和上边界，而“大西洋”处于大陆的右边界和下边界。

规定水流只能按照上、下、左、右四个方向流动，且只能从高到低或者在同等高度上流动。

请找出那些水流既可以流动到“太平洋”，又能流动到“大西洋”的陆地单元的坐标。

提示：

1. 输出坐标的顺序不重要
2. m 和 n 都小于150

示例：

给定下面的 5×5 矩阵：

太平洋 ~ ~ ~ ~ ~
~ 1 2 2 3 (5) *
~ 3 2 3 (4) (4) *
~ 2 4 (5) 3 1 *
~ (6) (7) 1 4 5 *
~ (5) 1 1 2 4 *
* * * * * 大西洋

返回：

`[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]]` (上图中带括号的单元).

跟 130 题被围绕的区域一样，不可能从中间入手，只能从周围入手。

找到从四周流往中心的路线，最终四条路线重叠的部分就是所求解。

也就是从低处往高处爬。

很明显，这个问题存在这大量的 dfs 叠加的问题，需要抽取出来。然后遍历即可。

结果超出时间限制，case 通过率 111/113

采用 queue 来做这种题目，visit 的赋值在取出的那一下赋值即可，方便简单。

```

class Solution:
    def pacificAtlantic(self, matrix: [[int]]) -> [[int]]:
        if len(matrix) == 0:
            return []
        M, N = len(matrix), len(matrix[0])
        visit1 = [[0] * N for _ in range(M)]
        visit2 = [[0] * N for _ in range(M)] # 只需两个数组，表示大西洋和太平洋即可
        res = []

        for i in range(M):
            self.dfs(matrix, i, 0, visit1)
            self.dfs(matrix, i, N - 1, visit2)
        for j in range(N):
            self.dfs(matrix, 0, j, visit1)
            self.dfs(matrix, M - 1, j, visit2) # 从四周的每一个位置dfs
        for i in range(M):
            for j in range(N):
                if visit1[i][j] == 1 and visit2[i][j] == 1: # 路径的交叉点满足题意
                    res.append([i, j])
        return res

    def dfs(self, matrix, x, y, visit):
        dx = [0, 1, 0, -1]
        dy = [1, 0, -1, 0]
        M, N = len(matrix), len(matrix[0])
        import collections
        queue = collections.deque()
        queue.append((x, y))
        while queue:
            x, y = queue.popleft()
            visit[x][y] = 1
            for i in range(4):
                nextX, nextY = x + dx[i], y + dy[i]
                if not (0 <= nextX < M and 0 <= nextY < N) or visit[nextX][nextY] == 1 or matrix[nextX][nextY] < matrix[x][y]:
                    continue
                if (nextX, nextY) not in queue:
                    queue.append((nextX, nextY))

```

1、迷宫 1\2\3\4

1、一个二维数组表示一个迷宫，0 表示为空，1 表示为路障，给定任一个位置，人是否可以走出迷宫，迷宫出口默认在二维数组右下角：

这种整型矩阵可以不设置 visit 数组，可直接赋值设置不回头。

Queue 其实也是回溯，只是因为 queue 因为先进先出的特性自带回溯。

Queue 的缺点就是无法求出具体路线。

```

class Solution(object):
    def dfs(self, start, destination, maze):
        if len(maze) == 0:
            return False
        M, N = len(maze), len(maze[0])
        visit = [[sys.maxsize] * N for _ in range(M)]
        dx = [0, 1, 0, -1]
        dy = [1, 0, -1, 0]
        import collections
        queue = collections.deque()
        queue.append(start)
        while queue:
            x, y = queue.popleft()
            maze[x][y] = 1 # 替代了visit数组
            if x == destination[0] and y == destination[1]:
                return True
            for i in range(4):
                nextX, nextY = x + dx[i], y + dy[i]
                if not (0 <= nextX < M and 0 <= nextY < N) or maze[nextX][nextY] == 1:
                    continue
                if (nextX, nextY) not in queue:
                    queue.append((nextX, nextY))
        return False

```

2、相比于 1，要找出走出迷宫的最短距离。

增设一个让循环进入 continue 的条件即可：

即：给经过的网格标数，当下一个网格的数不大于当前网格的数，表明你没有资格走到那个网格。

如何设置数呢，其实很简单：room[nextX][nextY] = room[x][y] + 1

而且这种题明显都是不需要 visit 数组的。

只要满足一下三个条件：

1、queue 中存的都是为 0 的元素坐标

2、障碍物为-1

3、可通过的地方为无限大

就可以通过 room[nextX][nextY] < room[x][y] + 1 进行排除很多坐标。

将 0 变为无穷大 , 1 变为 -1 , 初始位置为 1 , 当弹出的元素坐标为出口时 , 返回 room[x][y]

3、给定三个值初始化的 m 行 n 列的二维网格 , -1 表示墙壁或障碍物 , 0 表示门 , INF 表示空空间 , 使用值 2 的 31 次方 $-1 = 2147483647$ 表示 INF , 假设到门的距离小于 2147483647 。在代表每个空房间的网络中填入与门最近的距离 , 如果不可能到达门口 , 则应填入 INF 。

这道题就完美匹配上来 , queue 进行求最短路径的问题了 , 将所有的门先加入 queue 中 , 然后在往 queue 中添加空房间 , 最后如上题一样加一个下一个房间的数字要大于比当前房间的数字 + 1 才往下一个房间走的限制 , 即可。

```
class Solution:
    def wallsAndGates(self, rooms):
        if len(rooms) == 0:
            return []
        M, N = len(rooms), len(rooms[0])
        dx = [0, 1, 0, -1]
        dy = [1, 0, -1, 0]
        import collections
        queue = collections.deque()
        for i in range(M):
            for j in range(N):
                if rooms[i][j] == 0:
                    queue.append((i, j)) # 初始化queue
        while queue:
            x, y = queue.popleft()
            for i in range(4):
                nextX, nextY = x + dx[i], y + dy[i]
                if not (0 <= nextX < M and 0 <= nextY < N) or rooms[nextX][nextY] <= rooms[x][y] + 1:
                    continue
                queue.append((nextX, nextY))
                rooms[nextX][nextY] = rooms[x][y] + 1
        return rooms

# 主函数
if __name__ == '__main__':
    INF = 2147483647
    matrix = [[INF, -1, 0, INF],
              [INF, INF, INF, -1],
              [INF, -1, INF, -1],
              [0, -1, INF, INF]]
    solution = Solution()
    print("运行的结果是:", solution.wallsAndGates(matrix))
```

4、给定一个 m 行 n 列的二维字符数组表示迷宫。它有四种房间 , ‘S’ 代表从哪开

始（只有一个起点），‘E’ 代表迷宫的出口（当抵达出口时，将离开迷宫，该题目会有多个出口），‘*’ 代表这个房间可以经过，‘#’ 代表一堵墙，不能经过墙。每次可以上下左右移动到达一个房间，花费一分钟时间，但是不能到达墙。本例将得出离开这个迷宫所需的最少时间，如果不能离开，则返回-1.

将‘S’ --> 0, ‘E’ --> 0, * --> INF, # --> -1。并将所有的 E 的位置封装到 exit 列表中，当弹出的位置有一个在其中，那么就说明到了第一个出口，返回即可。

```

class Solution:
    def wallsAndGates(self, rooms):
        if len(rooms) == 0:
            return []
        M, N = len(rooms), len(rooms[0])
        dx = [0, 1, 0, -1]
        dy = [1, 0, -1, 0]
        import collections
        queue = collections.deque()
        exit = []
        import sys
        for i in range(M):
            for j in range(N):
                if rooms[i][j] == '*':
                    rooms[i][j] = sys.maxsize
                if rooms[i][j] == '#':
                    rooms[i][j] = -1
                if rooms[i][j] == 'S':
                    queue.append((i, j))
                    rooms[i][j] = 0
                if rooms[i][j] == 'E':
                    exit.append((i, j))
                    rooms[i][j] = sys.maxsize
        while queue:
            x, y = queue.popleft()
            if (x, y) in exit:
                return rooms[x][y]
            for i in range(4):
                nextX, nextY = x + dx[i], y + dy[i]
                if not (0 <= nextX < M and 0 <= nextY < N) or rooms[nextX][nextY] <= rooms[x][y] + 1:
                    continue
                queue.append((nextX, nextY))
                rooms[nextX][nextY] = rooms[x][y] + 1
        return -1

```

2、不同岛屿的个数 1\2

给定一个由 0 和 1 组成的非空二维网格代表一个岛屿，一个岛屿是指四个方向（包括

恒向和纵向)都相连的一组 1, 其中 1 表示陆地, 0 表示水域。假设网格的四个边缘都被水包围了, 找出所有不同岛屿的个数, 如果一个岛屿可以被转换(不考虑旋转和翻折)成另一个岛屿, 则认为两个岛屿是相同的。

关键在于如何判断两个岛屿是相同的

首先肯定进行的是 for 循环, 都是用 queue 进行遍历, 那么判断岛屿形状相同, 就用出 queue 的最后一个位置和出 queue 的第一个位置的位置之差是否相等即可。记得你算出一个岛的形状之后, 记得让这个岛消失。

特别要注意的是这里是三层循环, 那么注意用 i,j,k, 而不是用 i,j,i。

```

import collections
class Solution:
    def numberOfDistinctIslands(self, grid):
        if len(grid) == 0:
            return 0
        M, N = len(grid), len(grid[0])
        dx = [0, 1, 0, -1]
        dy = [1, 0, -1, 0]
        res = []
        for i in range(M):
            for j in range(N):
                if grid[i][j] == 1:
                    queue = collections.deque()
                    queue.append((i, j))
                    while queue:
                        x, y = queue.popleft()
                        grid[x][y] = 0 # 已经经过的1或0，这是关键
                        for k in range(4):
                            nextX, nextY = x + dx[k], y + dy[k]
                            if not (0 <= nextX < M and 0 <= nextY < N) or grid[nextX][nextY] == 0:
                                continue
                            if (nextX, nextY) not in queue:
                                queue.append((nextX, nextY))
                            if not queue: # 当queue为空时，表明(x, y)是最后一个元素
                                path = (x - i, y - j)
                                if path not in res:
                                    res.append(path)
                    print(grid)
                    return res

if __name__ == '__main__':
    grid = [
        [1, 1, 0, 0, 0],
        [1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1],
        [0, 0, 0, 1, 1]
    ]
    print("矩阵是:", grid)
    solution = Solution()
    print("不同岛屿个数是:", solution.numberOfDistinctIslands(grid))

```

4、能否到达终点\

给定一个 m 行 n 列的矩阵表示地图，1 代表空地，0 代表障碍物，9 代表终点，判断从 $(0,0)$ 开始能否到达终点，若能到达终点返回 True，否则返回 False

5、建立邮局

给出一个二维网格，每一格可以代表墙（2）、房子（1）和空（0），在网格中找到一

个空的位置建立邮局，使得所有房子到邮局的距离和是最小的。返回所有房子到邮局的最小

距离和，如果没有地方建立邮局，则返回-1。

这道题感觉和不同岛屿的个数的题目差不多，只不过要新建一个 SUM 数组，存储着房子到该处的距离和，再来一个 minDistanceArr 来计算房子到每个地方的最短距离。记得这个 minDistanceArr 的初始化是，障碍物为-1，空白处为无限大，起始处为 0，这也是最适合 queue 的，但记得要留个入口出来，不然就会一直 continue 下去

其中有墙和有房子的地方不能建造邮局，但是有房子的地方是可以通过的。

注意浅拷贝后操作原数组和所得数组会相互影响，只有深拷贝不会，
copy.deepcopy()。

```
class Solution:
    def shortestDistance(self, grid):
        rowNum = len(grid)
        colNum = len(grid[0])
        sum = [[0] * colNum for _ in range(rowNum)]
        for i in range(rowNum):
            for j in range(colNum):
                if grid[i][j] == 1:
                    visited = [[0] * colNum for _ in range(rowNum)] # visit[x][y]表示(i, j)到(x, y)的最短距离
                    self.dfs(grid, visited, sum, i, j)
        return sum

    def dfs(self, grid, visited, sum, i, j):
        dx = [0, 1, 0, -1]
        dy = [1, 0, -1, 0]
        queue = deque()
        queue.append((i, j))
        while queue:
            x, y = queue.popleft()
            for i in range(4):
                nextX = x + dx[i]
                nextY = y + dy[i]
                if not (0 <= nextX < len(grid) and 0 <= nextY < len(grid[0])) or visited[nextX][nextY] != 0 or \
                   grid[nextX][nextY] == 2: # 还要不能用 visited[nextX][nextY] < visited[x][y] + 1 因为障碍物不是-1，可以通过邮局才表示房子
                    continue
                queue.append((nextX, nextY))
                visited[nextX][nextY] = visited[x][y] + 1
                sum[nextX][nextY] += visited[nextX][nextY]
```

回溯递归

组合总和

39. 组合总和

难度 中等 777 收藏 分享 切换为英文 关注 反馈

给定一个无重复元素的数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中所有可以使数字和为 `target` 的组合。

`candidates` 中的数字可以无限制重复被选取。

说明：

- 所有数字（包括 `target`）都是正整数。
- 解集不能包含重复的组合。

示例 1：

```
输入：candidates = [2,3,6,7], target = 7,
所求解集为：
[
    [7],
    [2,2,3]
]
```

示例 2：

```
输入：candidates = [2,3,5], target = 8,
所求解集为：
[
    [2,2,2,2],
    [2,3,3],
    [3,5]
]
```

提示：

- `1 <= candidates.length <= 30`
- `1 <= candidates[i] <= 200`
- `candidate` 中的每个元素都是独一无二的。
- `1 <= target <= 500`

```

class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        N, res = len(candidates), []
        if N == 0:
            return []
        candidates.sort()
        self.backTrack(candidates, 0, [], res, target)
        return res

    def backTrack(self, candidates, start, tmp, res, target):
        N = len(candidates)
        if target == 0:
            res.append(tmp[:])
            return
        if target < 0:
            return
        for i in range(start, N): # 每次保持第一位可用，那么就能够确保每位重复使用
            if target < candidates[i]: # 上面的排序，就是为了这里的剪枝
                break
            self.backTrack(candidates, i, tmp + [candidates[i]], res, target - candidates[i])

```

组合总和 II

40. 组合总和 II

难度 中等 击 320 收藏 分享 切换为英文 关注 反馈

给定一个数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中所有可以使数字和为 `target` 的组合。

`candidates` 中的每个数字在每个组合中只能使用一次。

说明：

- 所有数字（包括目标数）都是正整数。
- 解集不能包含重复的组合。

示例 1：

```

输入: candidates = [10,1,2,7,6,1,5], target = 8,
所求解集为:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]

```

示例 2：

```

输入: candidates = [2,5,2,1,2], target = 5,
所求解集为:
[
  [1,2,2],
  [5]
]

```

```
class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        N, res = len(candidates), []
        if N == 0:
            return []
        candidates.sort()
        self.backTrack(0, [], res, candidates, target)
        return res

    def backTrack(self, start, tmp, res, candidates, target):
        N = len(candidates)
        if target == 0:
            if tmp not in res: # 这里可以用，因为数组里有重复元素
                res.append(tmp[:])
        if target < 0:
            return
        for i in range(start, N):
            if target < candidates[i]: return
            self.backTrack(i + 1, tmp + [candidates[i]], res, candidates, target - candidates[i])
```

组合总和 III

216. 组合总和 III

难度 中等 收藏 分享 切换为英文 关注 反馈

找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。

说明：

- 所有数字都是正整数。
- 解集不能包含重复的组合。

示例 1：

```
输入: k = 3, n = 7
输出: [[1,2,4]]
```

示例 2：

```
输入: k = 3, n = 9
输出: [[1,2,6], [1,3,5], [2,3,4]]
```

```
class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        if n < 6:
            return []
        res = []
        self.backTrack([], res, n, k, 1)
        return res

    def backTrack(self, tmp, res, n, k, start):
        if k < 0:
            return
        if k == 0 and n == 0:
            res.append(tmp[:])
        for i in range(start, 10):
            if n < i: # 因为这样 n - i < 0
                return
            self.backTrack(tmp + [i], res, n - i, k - 1, i + 1)
```

十以内的排列（阿里）

给一个 $n \leq 10$ ，按字典序输出 $1 \dots n$ ，相邻数字绝对值不为 1，的排列。

```
1 | 输入：
2 | 4
3 | 输出：
4 | 2 4 1 3
5 | 3 1 4 2
```

```

def f(vst, cur, N):
    if N == 0:
        print(" ".join(map(str, cur))) # 打印整形数组
    for i in range(1, len(vst)):
        if vst[i] or (len(cur) > 0 and i in [cur[-1]-1, cur[-1]+1]):
            continue
        vst[i] = 1
        f(vst, cur + [i], N - 1)
        vst[i] = 0

while True:
    N = input()
    if N == "":
        break
    N = int(N)
    f([0] * (N + 1), [], N)

```

全排列

46. 全排列

难度 中等 755 收藏 分享 切换为英文 关注 反馈

给定一个 没有重复 数字的序列，返回其所有可能的全排列。

示例：

```

输入: [1,2,3]
输出:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]

```

关键思想时，回溯数组， k 左边为已选集合， k 右边为候选集合，当 k 到达末尾，候选集合为空时，实现一次全排列，记录所有情况即可。

交换表示从候选集合中选择一个元素到已选集合后面，也就是第 k 个位置，可以说，每个元素都有可能在第 k 个位置。

所以出口是 $k == \text{len}(\text{nums})$ ，表明选择了 k 次。

```

class Solution:
    def permute(self, nums: [int]) -> [[int]]:
        res = []
        self.backTrack(nums, 0, res)
        return res

    def backTrack(self, nums, k, res): # 从k开始从候选集合中选择元素
        if k == len(nums): # 当选择的元素数量达到给定数组的长度时，表明没有了候选元素
            res.append(nums[:])
            return
        for i in range(k, len(nums)): # k之后的元素都是候选元素
            nums[i], nums[k] = nums[k], nums[i]
            self.backTrack(nums, k + 1, res)
            nums[i], nums[k] = nums[k], nums[i]

```

变种：1、从 n 个元素中选 m 个数全排列，那么就是已选集合达到 m 个时，记录一下即可。

```

class Solution:
    def permute(self, nums: [int], m) -> [[int]]:
        res = []
        self.backTrack(nums, 0, res, m)
        return res

    def backTrack(self, nums, k, res, m): # 从k开始从候选集合中选择元素
        if k == m: # 当选择的元素数量达到指定长度时，为一种情况
            res.append(nums[:m])
            return
        for i in range(k, len(nums)): # k之后的元素都是候选元素
            nums[i], nums[k] = nums[k], nums[i]
            self.backTrack(nums, k + 1, res, m)
            nums[i], nums[k] = nums[k], nums[i]

```

组合

77. 组合

难度 中等 290 收藏 分享 切换为英文 关注 反馈

给定两个整数 n 和 k ，返回 $1 \dots n$ 中所有可能的 k 个数的组合。

示例：

输入： $n = 4, k = 2$

输出：

[

[2,4],

[3,4],

[2,3],

[1,2],

[1,3],

[1,4],

]

```
class Solution:
    def combine(self, n: int, m: int) -> [[int]]:
        nums = []
        for i in range(1, n + 1):
            nums.append(i)
        res = []
        self.backtrack(nums, res, 0, m, [])
        return res

    def backtrack(self, nums, res, k, m, temp): # 取出m个数的组合
        if len(temp) == m: # 这里不再用 k == m 作为条件
            res.append(temp[:])
        for i in range(k, len(nums)):
            temp.append(nums[i]) # 不需要使用交换
            self.backtrack(nums, res, i + 1, m, temp) # 与排列不同的地方，这里是i+1,不是k+1
            temp.pop()
```

返回按照数组相对位置相同的元素的全排列，就是组合，不会存在重复。

那么我们就不需要交换了，那就也不需要严格按照 $k+1$ 去遍历位置。

只需要进行普通的回溯就好。

利用 `permutation` 和 `combinations` 函数进行全排列和组合：

```
from itertools import combinations, permutations

words = ["area", "lead", "wall"]
results = list(combinations(words, 2)) # 在数组中取两个数的所有可能情况，不分先后
print(results)
results2 = list(permutations(words, 2)) # 在数组中取两个数的所有排列情况，分先后
print(results2)

"D:\Program Files\Python37\python.exe" F:/PycharmProjects/算法/Python算法指南/回溯递归/_init__.py
[('area', 'lead'), ('area', 'wall'), ('lead', 'wall')]
[('area', 'lead'), ('area', 'wall'), ('lead', 'area'), ('lead', 'wall'), ('wall', 'area'), ('wall', 'lead')]
```

复原 IP 地址

93. 复原IP地址

难度 中等 286 收藏 分享 切换为英文 关注 反馈

给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。

有效的 IP 地址正好由四个整数（每个整数位于 0 到 255 之间组成），整数之间用 “.” 分隔。

示例：

```
输入: "25525511135"
输出: ["255.255.11.135", "255.255.111.35"]
```

思路：dfs (s , ans , result) 复原 s 的 ip 地址，并存入 ans 中，当 s 为空时，将 ans 存入 result。

```

class Solution:
    def restoreIpAddresses(self, s):
        ans, result, final = [], [], []
        self.dfs(s, ans, result)
        for x in result:
            final.append('.'.join(x))
        return final

    def dfs(self, s, ans, result):
        if len(s) > 16:
            return
        if s == "":
            if len(ans) == 4:
                if ans not in result:
                    result.append(ans[:])
            return
        for i in range(1, 4):
            integer = int(s[:i])
            if integer > 255 or (s[:i] != '0' and s[:i][0] == '0'): # ip地址每段开始不能是0
                continue
            self.dfs(s[i:], ans + [s[:i]], result)

```

括号生成

22. 括号生成

难度 中等 1127 收藏 分享 切换为英文 关注 反馈

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。

示例：

```

输入：n = 3
输出：[
    "((()))",
    "(()())",
    "(())()",
    "()(())",
    "()()()"
]
```

这是一道枚举回溯的题目，就是回溯的元素并不是那么有规律，不能用循环进行回溯。

1、函数规划： $f(\text{self}, l, r, \text{item}, \text{res})$ ， l 为左括号数， r 为右括号数， item 为一种有效的括号组合， res 存储所有的括号组合。

2、Return case :if r 小于 l: return ,当右括号数小于左括号数时，表明先放了右括号，

将无法组成有效括号。

3、Base case : if l == 0 or r == 0 : res.append(item[:]) return

4、回溯过程：

```
If l > 0: self.f(l - 1, r, item + '(', res)
If r > 0: self.f(l, r - 1, item + ')', res)
```

```
class Solution:
    def helper(self, l, r, item, res):      # 当左括号数为l，右括号数为r时，生成所有有效的括号组合
        if r < l:                      # 右括号的数不能少于左括号的数
            return
        if l == 0 and r == 0:
            res.append(item)
            return
        if l > 0:
            self.helper(l - 1, r, item + '(', res)  # 先放左括号
        if r > 0:
            self.helper(l, r - 1, item + ')', res)  # 先放右括号，两者没有先后关系，是并列的

    def generateParenthesis(self, n):
        if n == 0:
            return []
        res = []
        self.helper(n, n, "", res)
        return res
```

翻转游戏

给定一个只包含两种字符+和-的字符串,两个人轮流翻转“++”变成“--”。当一个人无法采取行动时游戏结束,另一个人将是赢家,本例将判断能否保证先手胜利。

我的思路:先遍历的是不同的‘++’的起始位置,然后换成‘--’后,再当做一个新字

符串翻转“++”,所以回溯的是不同‘++’的位置,定义一个 count,翻转一次,自加一次,当没有“++”后,如果 count 是奇数,说明能保证先手胜利。

```

class Solution:
    def canWin(self, s, count, flags):
        if not s.__contains__("++"):
            if count % 2 == 1:
                return True
            else:
                return False
        for i in range(len(s) - 1):
            tempS = s
            if s[i: i + 2] == "++": # 直到找到为止
                s = s[0: i] + "--" + s[i + 2:]
                self.canWin(s, count + 1, flags)
            s = tempS

```

电话号码的字母组合

17. 电话号码的字母组合

难度 中等 772 收藏 分享 切换为英文 关注 反馈

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。



示例:

输入："23"
输出：["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

说明:

尽管上面的答案是按字典序排列的，但是你可以任意选择答案输出的顺序。

```
class Solution(object):
    def letterCombinations(self, digits: str) -> [str]:
        if len(digits) == 0:
            return []
        results = []
        self.backtrack(digits, "", results)
        return results

    def backtrack(self, digits, ans, results): # 输入digits, 单一情况存入ans, 所有情况存入results, 因此ans
        jianpan = ["", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"]
        if len(digits) == 0:
            results.append(ans)
            return
        num = int(digits[0])
        size = len(jianpan[num])
        for i in range(size): # 回溯的是每一个数字对应的键位的字母
            self.backtrack(digits[1:], ans + jianpan[num][i], results)
```

整数替换

397. 整数替换

难度 中等 60 收藏 分享 切换为英文 关注 反馈

给定一个正整数 n ，你可以做如下操作：

1. 如果 n 是偶数，则用 $n / 2$ 替换 n 。
 2. 如果 n 是奇数，则可以用 $n + 1$ 或 $n - 1$ 替换 n 。
- n 变为 1 所需的最小替换次数是多少？

示例 1:

输入:

8

输出:

3

解释:

8 -> 4 -> 2 -> 1

示例 2:

输入:

7

输出:

4

解释:

7 -> 8 -> 4 -> 2 -> 1

或

7 -> 6 -> 3 -> 2 -> 1

```

class Solution:
    def integerReplacement(self, n):
        self.len = sys.maxsize
        self.dfs(n, [n])
        return self.len - 1

    def dfs(self, n, temp): # temp记录递进情况，以便记录替换次数
        if n == 1:
            self.len = min(self.len, len(temp))
            return
        if n % 2 == 0:
            temp.append(n // 2)
            self.dfs(n // 2, temp) # 一种情况，不存在回溯
        else:
            for x in [-1, 1]: # 考虑n+1和n-1两种情况
                self.dfs(n + x, temp + [n + x])

```

寻找丢失的数

给一个由 1 - n 的整数随机组成的一个字符串序列，其中丢失了一个整数，本例将找到它。比如:n = 20, str = "19201234567891011121314151618" 丢失的数是 17。

解题思路:该题拿到手的时候明显该使用回溯的方法,因为你无法判断你从左往右找的数字是一位还是两位,所以要根据一位两位进行回溯,那么出口的话,一是找到了 0,二是找到了重复的数字,三是到底了。

```

class Solution:
    def findMissing2(self, n, str):
        used = [False for _ in range(n + 1)]
        return self.find(n, str, used)

    def find(self, n, str, used): # 从index处开始找丢失的整数
        result = []
        if str == "":
            for i in range(1, n + 1):
                if not used[i]:
                    result.append(i)
            return result[0] if len(result) == 1 else -1 # 找到后，返回target
        if str[0] == '0':
            return -1 # 为0时直接返回
        for i in range(1, 3):
            num = int(str[:i])
            if not (1 <= num <= n) or used[num]:
                continue
            used[num] = True
            target = self.find(n, str[i:], used)
            if target != -1:
                return target
            used[num] = False
        return -1 # 两种类型的字符串都不是回文

```

分割回文串

131. 分割回文串

难度 中等 307 收藏 分享 切换为英文 关注 反馈

给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。

返回 s 所有可能的分割方案。

示例:

```

输入: "aab"
输出:
[
  ["aa", "b"],
  ["a", "a", "b"]
]

```

```

class Solution:
    def partition(self, s):
        result = []
        self.backtrack(s, [], result)
        return result

    def backtrack(self, s, temp, result):
        if s == "": # s[N:N]
            result.append(temp[:])
            return
        N = len(s)
        for i in range(N):
            if not self.isHuiwen(s[:i + 1]): # s[0:0] == ""
                continue
            self.backtrack(s[i + 1:], temp + [s[:i + 1]], result)

    def isHuiwen(self, s):
        return s[::-1] == s

```

八皇后

八皇后问题就是要考察你对回溯法的利用，基础思路为：

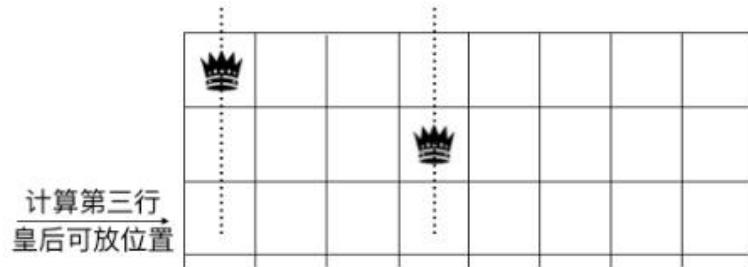
你可以在每一行放置一个皇后，放置的位置取决于上面的行放置的皇后的位置，如果没有摆放的位置后，那么就需要回溯，那么本题的难点在于，如何找到皇后能够拜访的位置，这个时候位运算就能解决这个烦恼。

Column 表示当前行的某个位置的该列有无皇后

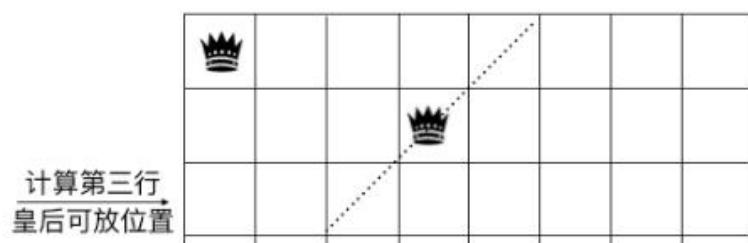
Pie 表示当前行的某个位置的右斜线有无皇后

Na 表示当前行的某个位置的左斜线有无皇后

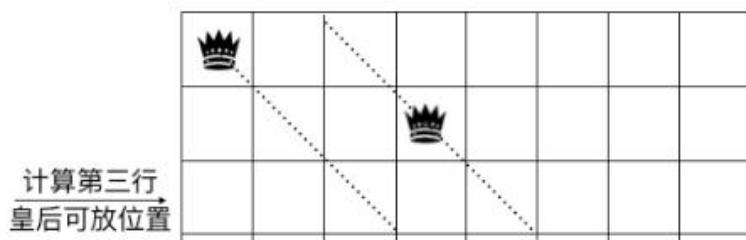
已放皇后对应的列均不可放皇后



已放皇后对应的左斜线经过的单元格均不可放皇后



已放皇后对应的右斜线经过的单元格均不可放皇后



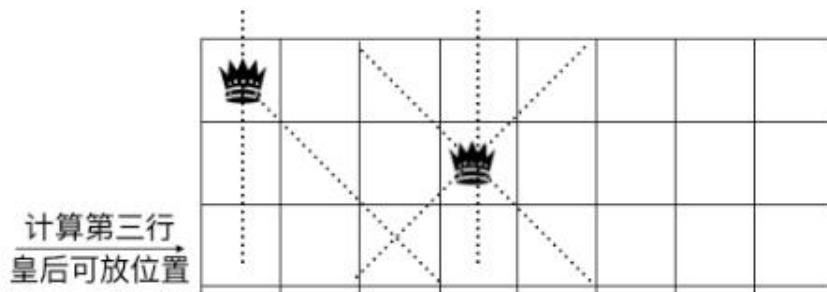
我们以 column 来记录所有上方行已放置的皇后导致当前行格子不可用的集合，所在列如果放了皇后，则当前行格子对应的位置为 1，否则为 0，同理，以 pie（撇，左斜线）记录所有已放置的皇后左斜方向导致当前行格子不可用的集合，na（捺，右斜线）表示所有已放置的皇后右斜方向导致当前行不可用的集合。则对于第三行来说我们有：

```
column = 10010000 (上图中的第一个图, 第 1, 4 列放了皇后, 所以 1, 4 位置为 1, 其他位置为 0)
pie = 00100000 (上图中的第二个图, 左斜线经过第三行的第三个方格, 所以第三位为 1)
na = 00101000 (上图中的第三个图, 右斜线经过第三行的第三, 五个方格, 所以第三, 五位为 1)
```

将这三个变量作或运算得到结果如下

```
10010000  
| 00100000  
| 00101000  
-----  
10111000
```

也就是说对于第三层来说第 1, 3, 4, 5 个格子不能放皇后。如图示



于是可知 `column | pie | na` 得到的结果中值为 0 的代表当前行对应的格子可放皇后，1 代表不能放，但我们想改成 1 代表格子可放皇后，0 代表不可放皇后，毕竟这样更符合我们的思维方式，怎么办，取反不就行了，于是我们有`~(column | pie | na)`。

问题来了，这样取反是有问题的，因为这三个变量都是定义的 int 型，为 32 位，取反之后高位的 0 全部变成了 1，而我们只想保留低 8 位（因为是 8 皇后），想把高位都置为 0，怎么办，这里就要用到位运算的黑科技了

```
~(column | pie | na) & ((1 << 8)-1)
```

后面的的 `((1<< 8)-1)` 表示先把 1 往左移 8 位，值为 100000000,再减 1，则低 8 位全部为 1，高位全部为 0！（即 001111111）再作与运算，即可保留低 8 位，去除高位。

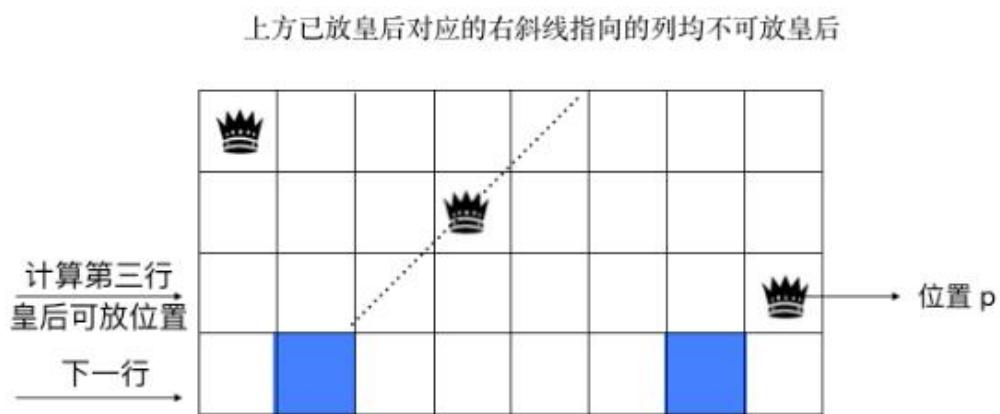
后面的的 $((1 << 8) - 1)$ 表示先把 1 往左移 8 位，值为 100000000，再减 1，则低 8 位全部为 1，高位全部为 0！（即 001111111）再作与运算，即可保留低 8 位，去除高位。

费了这么大的劲，我们终于把当前行可放皇后的格子都找出来了（所有位值为 1 的格子可放置皇后）。接下来我们只要做个循环，遍历所有位为 1 的格子，每次取出可用格子放上皇后，再找下一层可放置皇后的格子，依此递归下去即可，直到所有行都遍历完毕（递归的终止条件）。

还有一个问题，已知当前行的 column, pie, na，怎么确定下一行的 column, pie, na 的值（毕竟选完当前行的皇后后，要确定下一行的可用格子，而下一行的可用格子依赖于 column, pie, na 的值）

上文可知，我们已经选出了当前行可用的格子（相应位为 1 对应的格子可用），假设我们在当前行选择了其中一个格子来放置皇后，此位置记为 p（如果是当前行的最后一个格子最后一个格子，则值为 1，如果放在倒数第二个，值为 10，倒数第三个则为 100，依此类推），则对于下一行来说，显然 $\text{column} = \text{column} | p$

那么 pie 呢，仔细看下图，显然应该为 $(\text{pie} | p) << 1$ ，左斜线往下一行的格子延展时，相当于左移一位，



同理下一行的 na 为 $(\text{na} | p) >> 1$ 。

将三者想或即得到了该行能够放置皇后的位置 pos。

那么放置过后如何能够确认下一行能够放置皇后的位置呢？

假设你放置的皇后的位置是 p，那么你将 pos 的 p 位置之外的位置置 0 即可，然后你用 $p|\text{column}$ 得到新的 column，用 $p|\text{pie} << 1$ 得到新的 pie，用 $p|\text{na} >> 1$ 得到新的 na。

相关伪代码如下：

```
void queenSettle(row, column,pie,na) {
    N = 8; // 8皇后
    if (row >= N) {
        // 遍历到最后一行说明已经找到符合的条件了
        count++;return
    }

    // 取出当前行可放置皇后的格子
    bits = (~column|pie|na) & ((1 << N)-1)

    while(bits > 0) {
        // 每次从当前行可用的格子中取出最右边位为 1 的格子放置皇后
        p = bits & -bits

        // 紧接着在下一行继续放皇后
        queenSettle(row+1, column | p, (pie|p) << 1, (na | p) >> 1)

        // 当前行最右边格子已经选完了，将其置成 0，代表这个格子已遍历过
        bits = bits & (bits-1)
    }
}
```

一开始传入 queenSettle(0,0,0,0) 这样即可得到最终的解。伪代码写得很清楚了，相信用相关语言不难实现，这里就留给大家作个练习吧。

这道题的难点在于如何设计回溯的条件。

因为 bits &-bits 能找到最右边为 1 的格子

且 bits & bits - 1 能将最右边为 1 的格子置 0.

所以 bits 中的 1 代表的必须是能够放置皇后的位置 p。

但是 column\pie\na 中的 1 代表的是放置了皇后的位置，也就是不能放皇后的位置。

所以 p|column 得到的 1 是不能放置皇后的位置，所以需要取反。

```

class Solution:
    count = 0
    def solveNQueens(self, N: int) -> [[str]]:
        self.queenSettle(N, 0, 0, 0)
        return self.count

    def queenSettle(self, N, row, column, pie, na):
        if row >= N:
            self.count += 1
            return
        bits = (~column | pie | na) & ((1 << N) - 1) # 为1的位置可放置皇后
        while bits > 0: # 当为0时表明没有地方可以放置皇后，需要回溯
            p = bits & -bits # 每次从当前行可用的各自中取出最右边为1的格子放置皇后
            self.queenSettle(N, row + 1, column | p, (pie | p) << 1, (na | p) >> 1)
            bits = bits & (bits - 1) # 当前行的最右边各自已经选完了，将其置成0，代表这个格子已被遍历过。

```

面试题 08.12. 八皇后

在八皇后的基础上，只要将 p 用相应的字符串表达出来即可。

面试题 08.12. 八皇后

难度 困难 击 23 收藏 分享 切换为英文 关注 反馈

设计一种算法，打印 N 皇后在 $N \times N$ 棋盘上的各种摆法，其中每个皇后都不同行、不同列，也不在对角线上。这里的“对角线”指的是所有的对角线，不只是平分整个棋盘的那两条对角线。

注意：本题相对原题做了扩展

示例：

输入：4
输出：[[".Q..", "...Q", "Q...", "...Q."], ["..Q.", "Q...", "...Q", ".Q.."]]
解释：4 皇后问题存在如下两个不同的解法。

```
[
[".Q..", // 解法 1
 "...Q",
 "Q...",
 "...Q."],
["..Q.", // 解法 2
 "Q...",
 "...Q",
 ".Q.."]
]
```

```

class Solution:
    def solveNQueens(self, N: int) -> [[str]]:
        result = []
        self.queenSettle(N, 0, 0, 0, 0, [], result)
        return result

    def queenSettle(self, N, row, column, pie, na, temp, result):
        if row >= N:
            result.append(temp[:])
            return
        bits = (~column | pie | na) & ((1 << N) - 1) # 为1的位置可放置皇后
        while bits > 0: # 当为0时表明没有地方可以放置皇后，需要回溯
            p = bits & -bits # 每次从当前行可用的各自中取出最右边为1的各自放置皇后
            str = bin(p)[2:].replace('1', 'Q').replace("0", ".", N-1)
            if len(str) < N:
                str = "." * (N - len(str)) + str # 不满N位的补上小数点
            temp.append(str)
            self.queenSettle(N, row + 1, column | p, (pie | p) << 1, (na | p) >> 1, temp, result)
            temp.pop()
            bits = bits & (bits - 1) # 当前行的最右边格子已经选完了，表明该列不可再选。

```

递归

四数之和

18. 四数之和

难度 中等 533 收藏 分享 切换为英文 关注 反馈

给定一个包含 n 个整数的数组 nums 和一个目标值 target ，判断 nums 中是否存在四个元素 a, b, c 和 d ，使得 $a + b + c + d$ 的值与 target 相等？找出所有满足条件且不重复的四元组。

注意：

答案中不可以包含重复的四元组。

示例：

给定数组 $\text{nums} = [1, 0, -1, 0, -2, 2]$ ，和 $\text{target} = 0$ 。

满足要求的四元组集合为：

```
[
  [-1, 0, 0, 1],
  [-2, -1, 1, 2],
  [-2, 0, 0, 2]
]
```

Case pass : 192 / 282

```

class Solution:
    def fourSum(self, nums: [int], target: int) -> [[int]]:
        nums.sort()
        return self.nSum(nums, target, 4)

    def nSum(self, nums, target, n):
        if n == 2:
            return self.twoSum(nums, target)
        else:
            res = []
            for i in range(len(nums) - n + 1): # 留出后续n个数的位置
                temp = self.nSum(nums[i + 1:], target - nums[i], n - 1)
                for x in temp:
                    res.append([nums[i]] + x)
            return res

    def twoSum(self, nums, target): # 返回两数之和等于target的组合，所以不能用hash表，因为要去重
        N, res = len(nums), []
        left, right = 0, N - 1
        while left < right:
            if nums[left] + nums[right] > target:
                right -= 1
            elif nums[left] + nums[right] < target:
                left += 1
            else:
                while left + 1 < N and nums[left] == nums[left + 1]: # 因此nums需要时排序的，方便去重
                    left += 1
                while right - 1 >= 0 and nums[right - 1] == nums[right]:
                    right -= 1
                res.append([nums[left], nums[right]])
                left += 1
                right -= 1
        return res

```

汉诺塔问题

```

def hanoi(self, n, A, B, C): # 将n个盘子从A移动到C
    if n == 1:
        self.move(A, C)
        return
    self.hanoi(n - 1, A, C, B)
    self.move(A, C)
    self.hanoi(n-1, B, A, C)

def move(self, A, B):
    print(A + " -> " + B)

```

双指针

供暖器

475. 供暖器

难度 简单 146 收藏 分享 切换为英文 关注 反馈

冬季已经来临。你的任务是设计一个有固定加热半径的供暖器向所有房屋供暖。

现在，给出位于一条水平线上的房屋和供暖器的位置，找到可以覆盖所有房屋的最小加热半径。

所以，你的输入将会是房屋和供暖器的位置。你将输出供暖器的最小加热半径。

说明:

- 给出的房屋和供暖器的数目是非负数且不会超过 25000。
- 给出的房屋和供暖器的位置均是非负数且不会超过 10^9 。
- 只要房屋位于供暖器的半径内(包括在边缘上)，它就可以得到供暖。
- 所有供暖器都遵循你的半径标准，加热的半径也一样。

示例 1:

输入: [1, 2, 3], [2]

输出: 1

解释: 仅在位置2上有一个供暖器。如果我们将加热半径设为1，那么所有房屋就都能得到供暖。

示例 2:

输入: [1, 2, 3, 4], [1, 4]

输出: 1

解释: 在位置1, 4上有两个供暖器。我们需要将加热半径设为1，这样所有房屋就都能得到供暖。

思路:二分查找供暖器的位置,找到离房屋最近的两个供暖器,计算最近的距离,然后整体求房屋离供暖期最大的距离.下面是我自己的方法,但是无法通过所有的测试用例.

```
class Solution:
    def findRadius(self, houses: List[int], heaters: List[int]) -> int:
        houses.sort()
        heaters.sort()
        res = 0
        for house in houses:
            left, right = 0, len(heaters) - 1
            while left + 1 < right: # 二分查找可以用来查找最接近某个值的两个值
                mid = (left + right) // 2
                if heaters[mid] < house:
                    left = mid + 1
                else:
                    right = mid
            res = max(res, min(abs(house - heaters[left]), abs(heaters[right] - house)))
        return res
```

下面是大佬的代码:

```

class Solution:
    def findRadius(self, houses: List[int], heaters: List[int]) -> int:
        houses.sort()
        heaters.sort()
        res = 0
        # 每个房子被供暖需要的最小半径
        for house in houses:
            left = 0
            right = len(heaters) - 1
            while left < right:
                middle = left + (right - left) // 2
                if heaters[middle] < house:
                    left = middle + 1
                else:
                    right = middle

            # 供暖器和房子的位置相等
            if heaters[left] == house:
                house_res = 0
            # 此时是最接近房子的供暖器
            elif heaters[left] < house:
                house_res = house - heaters[left]
            # 供暖器不一定是最近房子的
            else:
                # 第一个供暖器
                if left == 0:
                    house_res = heaters[left] - house
                else:
                    # 这个供暖器和前一个供暖器哪个更接近
                    house_res = min(heaters[left] - house, house - heaters[left - 1])

            res = max(house_res, res)

        return res

```

寻找旋转排序数组中的最小值 I

153. 寻找旋转排序数组中的最小值

难度 中等 231 收藏 分享 切换为英文 关注 反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

示例 1:

输入: [3,4,5,1,2]
输出: 1

示例 2:

输入: [4,5,6,7,0,1,2]
输出: 0

```

class Solution:
    def findMin(self, nums: List[int]) -> int:
        left, right, target = 0, len(nums), nums[-1]
        if len(nums) == 1:
            return nums[0]
        while left < right:
            mid = (left + right) // 2
            if nums[mid] <= target:
                right = mid      # 这里不能-1，因为可能就是最小元素
            else:
                left = mid + 1 # 不等于的地方+1
        return nums[left]

```

寻找旋转排序数组中的最小值 II

154. 寻找旋转排序数组中的最小值 II

难度 困难 161 收藏 分享 切换为英文 关注 反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

请找出其中最小的元素。

注意数组中可能存在重复的元素。

示例 1：

输入: [1,3,5]
输出: 1

示例 2：

输入: [2,2,2,0,1]
输出: 0

说明：

- 这道题是 寻找旋转排序数组中的最小值 的延伸题目。
- 允许重复会影响算法的时间复杂度吗？会如何影响，为什么？

因为会重复，加入继续以最右边的数作为 target，那么 target 可能存在于首尾，所以这里定义一个动态 target，也就是 nums[right]，当遇到相等情况时，无法确定最小数在那边，所以 right -= 1。

```

class Solution:
    def findMin(self, nums: List[int]) -> int:
        N = len(nums)
        left, right = 0, N - 1
        while left < right:
            mid = (left + right) // 2
            if nums[mid] < nums[right]:
                right = mid
            elif nums[mid] > nums[right]:
                left = mid + 1
            else:
                right -= 1 # 因为可能出现重复，所以始终以最右边的不重复位置来判断
        return nums[left]

```

搜索旋转排序数组

33. 搜索旋转排序数组

难度 中等 822 收藏 分享 切换为英文 关注 反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 `-1`。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 $O(\log n)$ 级别。

示例 1：

```
输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4
```

示例 2：

```
输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1
```

二分查找最重要的就是要缩小范围，而当一个数组不是有序的时候，就很难确定范围，故需讨论 `nums[left]` 和 `nums[mid]` 的大小：

If `nums[left] < nums[mid]`: 表明 `nums[left:mid]` 是升序

If `nums[left] > nums[mid]`: 表明 `nums[mid:]` 是升序

```
class Solution:
    def search(self, nums: [int], target: int) -> bool:
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) // 2
            if nums[mid] == target:
                return mid
            if nums[left] <= nums[mid]: # 先要确定其在哪一段，才能进行正常的二分查找
                if nums[left] <= target < nums[mid]: # 确定的那段要具体范围
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
        return -1
```

搜索旋转排序数组 2

81. 搜索旋转排序数组 II

难度 中等 185 收藏 分享 切换为英文 关注 反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,0,1,2,2,5,6]` 可能变为 `[2,5,6,0,0,1,2]`)。

编写一个函数来判断给定的目标值是否存在与数组中。若存在返回 true，否则返回 false。

示例 1:

输入: nums = [2, 5, 6, 0, 0, 1, 2], target = 0
输出: true

示例 2：

输入: nums = [2,5,6,0,0,1,2], target = 3
输出: false

进阶：

- 这是 [搜索旋转排序数组](#) 的延伸题目，本题中的 `nums` 可能包含重复元素。
 - 这会影响到程序的时间复杂度吗？会有怎样的影响，为什么？

难点：无法根据 $\text{nums}[\text{left}]$ 和 $\text{nums}[\text{mid}]$ 来找到升序序列；

举例 : [1,1,1,1,3] --> [1,3,1,1,1], $\text{nums}[\text{left}] == \text{nusm}[\text{mid}]$, 但是[1,3,1]乱序

解决：当相等时， $\text{left} += 1$ ，直到不等

寻找峰值

162. 寻找峰值

难度 中等 245 收藏 分享 切换为英文 关注 反馈

峰值元素是指其值大于左右相邻值的元素。

给定一个输入数组 `nums`，其中 `nums[i] > nums[i+1]`，找到峰值元素并返回其索引。

数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。

你可以假设 `nums[-1] = nums[n] = -∞`。

示例 1：

```
输入: nums = [1, 2, 3, 1]
输出: 2
解释: 3 是峰值元素，你的函数应该返回其索引 2。
```

示例 2：

```
输入: nums = [1, 2, 1, 3, 5, 6, 4]
输出: 1 或 5
解释: 你的函数可以返回索引 1，其峰值元素为 2；或者返回索引 5，其峰值元素为 6。
```

说明：

你的解法应该是 $O(\log N)$ 时间复杂度的。

```
class Solution:
    def findPeakElement(self, nums: [int]) -> int:
        left, right = 0, len(nums) - 1
        while left < right:
            mid = (left + right) // 2
            if nums[mid] < nums[mid + 1]: # 相邻二分查找, 没有target值
                left = mid + 1
            else:
                right = mid
        return right
```

第一个错误的版本

278. 第一个错误的版本

难度 [简单](#) [189](#) [收藏](#) [分享](#) [切换为英文](#) [关注](#) [反馈](#)

你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最新版本没有通过质量检测。由于每个版本都是基于之前的版本开发的，所以错误的版本之后的所有版本都是错的。

假设你有 n 个版本 $[1, 2, \dots, n]$ ，你想找出导致之后所有版本出错的第一个错误的版本。

你可以通过调用 `bool isBadVersion(version)` 接口来判断版本号 `version` 是否在单元测试中出错。实现一个函数来查找第一个错误的版本。你应该尽量减少对调用 API 的次数。

示例：

给定 $n = 5$ ，并且 `version = 4` 是第一个错误的版本。

调用 `isBadVersion(3) -> false`
调用 `isBadVersion(5) -> true`
调用 `isBadVersion(4) -> true`

所以， 4 是第一个错误的版本。

```
class Solution:
    def firstBadVersion(self, n):
        left, right = 1, n
        while left < right:
            mid = (left + right) // 2
            if isBadVersion(mid) == False and isBadVersion(mid + 1) == True:
                return mid + 1
            elif isBadVersion(mid) == False and isBadVersion(mid + 1) == False:
                left = mid + 1
            else:
                right = mid
        return 1 # 一个元素时是无法二分的
```

x 的平方根

69. x 的平方根

难度 简单 收藏 分享 切换为英文 关注 反馈

实现 `int sqrt(int x)` 函数。

计算并返回 x 的平方根，其中 x 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1：

输入：4

输出：2

示例 2：

输入：8

输出：2

说明：8 的平方根是 2.82842...,

由于返回类型是整数，小数部分将被舍去。

```
class Solution:
    def mySqrt(self, x: int) -> int:
        l, r, ans = 0, x, -1
        while l <= r: # 这里必须是等号,因为l比ans大1,所以必须让ans能所有整数都能取到
            mid = (l + r) // 2
            if mid * mid <= x:
                ans = mid # 因为往下取整,所以应该取左边值
                l = mid + 1 # 这里需要 + 1,不然会陷入死循环,因为mid和l是相互影响的,必须有一个是变动的
            else:
                r = mid - 1
        return ans
```

求 `sqrt(2)`，要求精确到小数点后十位

思路：肯定是要用二分法，`left = 1.4, right = 1.5`，定义临界点 `point = 0.0000000001`，

当 `right - left = point` 时结束寻找。因为这里 `mid` 是除法符号，不是整除符号，所以 `l` 和 `r` 无所谓

```

if __name__ == '__main__':
    point = 0.0000000001
    left, right = 0, 2
    while right - left > point:
        mid = (left + right) / 2
        if mid * mid > 2:
            right = mid
        else:
            left = mid
    print(left)           1.4142135622911156
    print(mid)           1.4142135622911156
    print(right)         1.4142135623842478
    print(round(mid, 10)) 1.4142135623

```

爱吃香蕉的珂珂

875. 爱吃香蕉的珂珂

难度 中等 86 收藏 分享 切换为英文 关注 反馈

珂珂喜欢吃香蕉。这里有 N 堆香蕉，第 i 堆中有 $piles[i]$ 根香蕉。警卫已经离开了，将在 H 小时后回来。

珂珂可以决定她吃香蕉的速度 K （单位：根/小时）。每个小时，她将选择一堆香蕉，从中吃掉 K 根。如果这堆香蕉少于 K 根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。

珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。

返回她可以在 H 小时内吃掉所有香蕉的最小速度 K （ K 为整数）。

示例 1：

输入: piles = [3,6,7,11], H = 8
输出: 4

示例 2：

输入: piles = [30,11,23,4,20], H = 5
输出: 30

示例 3：

输入: piles = [30,11,23,4,20], H = 6
输出: 23

提示：

- $1 \leq piles.length \leq 10^4$
- $piles.length \leq H \leq 10^9$
- $1 \leq piles[i] \leq 10^9$

思路：用二分法取 K 值，看 K 值是否满足，这里计算多少小时吃完的方法非常巧妙：

$(p - 1) // K + 1$ ，既满足了正好为 K 的整数倍，也满足了不是的情况。

```

class Solution:
    def minEatingSpeed(self, piles: List[int], H: int) -> int:
        def possible(K):
            if K == 0:
                return False
            return sum((p - 1) // K + 1 for p in piles) <= H # 这里的p-1用的非常巧妙

        lo, hi = 0, max(piles)
        while lo < hi:
            mid = (lo + hi) // 2
            if possible(mid):
                hi = mid # 如果速度可以，那么就减点速度，但是这个时候不能-1，因为很有可能下一个数就没办法吃完了
            else:
                lo = mid + 1 # 如果速度慢了，就要加速了
        return lo

```

找到 k 个最接近的元素

658. 找到 K 个最接近的元素

难度 中等 119 收藏 分享 切换为英文 关注 反馈

给定一个排序好的数组，两个整数 k 和 x ，从数组中找到最靠近 x （两数之差最小）的 k 个数。返回的结果必须要是按升序排好的。如果有两个数与 x 的差值一样，优先选择数值较小的那个数。

示例 1：

输入: [1,2,3,4,5], k=4, x=3
输出: [1,2,3,4]

示例 2：

输入: [1,2,3,4,5], k=4, x=-1
输出: [1,2,3,4]

排除 $n - k$ 个元素，找到最接近的 k 个元素开头， $right - left == k$ 其实还剩下 $k + 1$ 个数，加 1 就会多舍弃一个数

```

class Solution:
    def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:
        N = len(arr)
        if N == 0:
            return []
        left, right = 0, N - 1
        while right - left + 1 > k: # 当还剩余k个数时跳出循环
            if x - arr[left] <= arr[right] - x:
                right -= 1
            else:
                left += 1
        return arr[left : left + k]

```

整数反转

7. 整数反转

难度 简单 2066 收藏 分享 切换为英文 关注 反馈

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

示例 1：

```
输入: 123
输出: 321
```

示例 2：

```
输入: -123
输出: -321
```

示例 3：

```
输入: 120
输出: 21
```

注意反转元素的方法

```
class Solution:
    def reverse(self, x: int) -> int:
        listX = [v for v in str(x)]
        temp = ""
        if listX[0] == '-':
            temp = "-"
            listX = listX[1:]
        left, right = 0, len(listX) - 1
        while left < right:
            listX[left], listX[right] = listX[right], listX[left]
            left += 1
            right -= 1
        strX = ''.join(listX)
        x = int(strX)
        res = int(temp + str(x)) # 需要加上符号化为整数,看是否在范围内
        return res if -math.pow(2, 31) <= res <= math.pow(2, 31) - 1 else 0
```

搜索二维矩阵

74. 搜索二维矩阵

难度 中等 214 收藏 分享 切换为英文 关注 反馈

编写一个高效的算法来判断 $m \times n$ 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

- 每行中的整数从左到右按升序排列。
- 每行的第一个整数大于前一行的最后一个整数。

示例 1：

```
输入:  
matrix = [  
    [1, 3, 5, 7],  
    [10, 11, 16, 20],  
    [23, 30, 34, 50]  
]  
target = 3  
输出: true
```

示例 2：

```
输入:  
matrix = [  
    [1, 3, 5, 7],  
    [10, 11, 16, 20],  
    [23, 30, 34, 50]  
]  
target = 13  
输出: false
```

```
class Solution:  
    def searchMatrix(self, matrix: [[int]], target: int) -> bool:  
        m = len(matrix)  
        if m == 0:  
            return False  
        n = len(matrix[0])  
        if n == 0:  
            return False  
        left, right = 0, m * n - 1  
        while left <= right: # 二分查找在里面的取值时，这里需加上等号  
            mid = (left + right) // 2 # mid是一维的  
            x = mid // n # mid 整除得 行  
            y = mid % n # mid 取余得 列  
            if target < matrix[x][y]:  
                right = mid - 1  
            elif target > matrix[x][y]:  
                left = mid + 1  
            else:  
                return True  
        return False
```

搜索二维矩阵 2

240. 搜索二维矩阵 II

难度 中等 342 收藏 分享 切换为英文 关注 反馈

编写一个高效的算法来搜索 $m \times n$ 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

示例：

现有矩阵 matrix 如下：

```
[  
    [1, 4, 7, 11, 15],  
    [2, 5, 8, 12, 19],  
    [3, 6, 9, 16, 22],  
    [10, 13, 14, 17, 24],  
    [18, 21, 23, 26, 30]  
]
```

给定 target = 5，返回 true。

给定 target = 20，返回 false。

这里不能用二分法，所以得用 while 循环，注意别忘了循环跳出条件.

```
class Solution:  
    def searchMatrix(self, matrix, target):  
        if len(matrix) == 0 or len(matrix[0]) == 0: # 不可或缺  
            return False  
        M, N = len(matrix), len(matrix[0])  
        x, y = 0, N - 1  
        cur = matrix[x][y]  
        while True:  
            if cur < target:  
                x += 1  
            elif cur > target:  
                y -= 1  
            else:  
                return True  
            if x >= M or y < 0:  
                break  
            cur = matrix[x][y]  
        return False
```

合并两个有序数组

88. 合并两个有序数组

难度 **简单** 点 559 收藏 分享 切换为英文 关注 反馈

给你两个有序整数数组 $nums1$ 和 $nums2$ ，请你将 $nums2$ 合并到 $nums1$ 中，使 $nums1$ 成为一个有序数组。

说明:

- 初始化 $nums1$ 和 $nums2$ 的元素数量分别为 m 和 n 。
- 你可以假设 $nums1$ 有足够的空间（空间大小大于或等于 $m + n$ ）来保存 $nums2$ 中的元素。

示例:

输入:
 $nums1 = [1, 2, 3, 0, 0, 0]$, $m = 3$
 $nums2 = [2, 5, 6]$, $n = 3$

输出: $[1, 2, 2, 3, 5, 6]$

新元素的位置记得 + 1，因为双指针都分别减了 1.

```
class Solution(object):
    def merge(self, nums1, m, nums2, n):
        M, N = m - 1, n - 1 # 从大到小进行合并
        while M >= 0 and N >= 0:
            if nums1[M] <= nums2[N]:
                nums1[M + N + 1] = nums2[N] # 在nums1的基础上合并
                N -= 1
            else:
                nums1[M + N + 1] = nums1[M]
                M -= 1
        while N >= 0: # M大于0的话，本身就在原位置， 不用动
            nums1[N] = nums2[N]
            N -= 1
        return nums1
```

合并区间

56. 合并区间

难度 中等 493 收藏 分享

给出一个区间的集合，请合并所有重叠的区间。

示例 1：

```
输入: [[1,3],[2,6],[8,10],[15,18]]
输出: [[1,6],[8,10],[15,18]]
解释: 区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].
```

示例 2：

```
输入: [[1,4],[4,5]]
输出: [[1,5]]
解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

先排序后合并

```
class Solution:
    def merge(self, intervals: [[int]]) -> [[int]]:
        if len(intervals) == 0:
            return []
        intervals.sort()
        result, N = [], len(intervals)
        result.append(intervals[0])
        for i in range(1, N):
            if intervals[i][0] <= result[-1][1]: # 首先明确上一个元素的右界大于下一个元素的左界后，就不能添加到result中
                if result[-1][1] <= intervals[i][1]: # 交叉关系，更新右界，包含关系，保持不变
                    result[-1][1] = intervals[i][1]
                else:
                    result.append(intervals[i])
        return result
```

两数之和 2

167. 两数之和 II - 输入有序数组

难度 简单 310 收藏 分享 切换为英文 关注 反馈

给定一个已按照升序排列的有序数组，找到两个数使得它们相加之和等于目标数。

函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

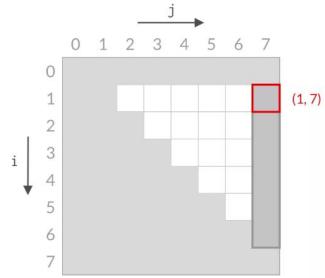
说明:

- 返回的下标值 (index1 和 index2) 不是从零开始的。
- 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

示例:

```
输入: numbers = [2, 7, 11, 15], target = 9
输出: [1,2]
解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。
```

当然可以采用两数之和的 hash 表的方法去做，但是因为是有序的，如果利用双指针去缩减搜索空间，这样才是最适合这道题的解法。



排除 $j=7$ 的全部解

```
class Solution:
    def twoSum(self, numbers: [int], target: int) -> [int]:
        N = len(numbers)
        if N < 2:
            return []
        left, right = 0, N - 1
        while left < right:
            if numbers[left] + numbers[right] < target:
                left += 1
            elif numbers[left] + numbers[right] > target:
                right -= 1
            else:
                return [left + 1, right + 1]
        return []
```

三数之和

15. 三数之和

难度 中等 2302 收藏 分享 切换为英文 关注 反馈

给你一个包含 n 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 a, b, c ，使得 $a + b + c = 0$ ？请你找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

给定数组 `nums = [-1, 0, 1, 2, -1, -4]`,

满足要求的三元组集合为：

```
[  
    [-1, 0, 1],  
    [-1, -1, 2]  
]
```

```
class Solution:  
    def threeSum(self, nums: [int]) -> [[int]]:  
        nums, N, res = sorted(nums), len(nums), []  
        for start in range(N):  
            if nums[start] > 0: # 因为已排好序，第一个大于0，那么不可能再等于0  
                return res  
            if start > 0 and nums[start] == nums[start - 1]: # 需要和上一次枚举的数不相同  
                continue  
            nextSum = 0 - nums[start]  
            left, right = start + 1, N - 1 # 对后面的数进行两数求和  
            while left < right:  
                if nums[left] + nums[right] == nextSum:  
                    while left + 1 < N and nums[left] == nums[left + 1]: # 去重  
                        left += 1  
                    while right + 1 < N and nums[right] == nums[right + 1]:  
                        right -= 1  
                    temp = [nums[start], nums[left], nums[right]]  
                    res.append(temp[:])  
                    left += 1  
                    right -= 1  
                elif nums[left] + nums[right] < nextSum:  
                    left += 1  
                else:  
                    right -= 1  
        return res
```

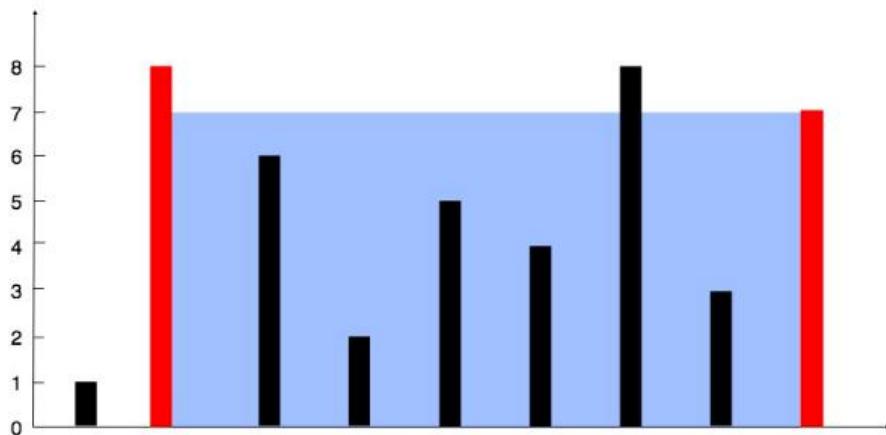
盛最多水的容器

11. 盛最多水的容器

难度 中等 1559 收藏 分享 切换为英文 关注 反馈

给你 n 个非负整数 a_1, a_2, \dots, a_n ，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器，且 n 的值至少为 2。



图中垂直线代表输入数组 $[1,8,6,2,5,4,8,3,7]$ 。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

很明显这道题就是要用到双指针的思想，也就是求最大面积，那么就是尽可能让底足够长，要么让高足够高，定义一个全局变量，实现 $O(n)$ 的时间复杂度。

而且一定是先移动较短的那个柱子，假如移动的是较长那端的柱子，那么水的高度不会增加，而且底却在一直减小，显然不合理。

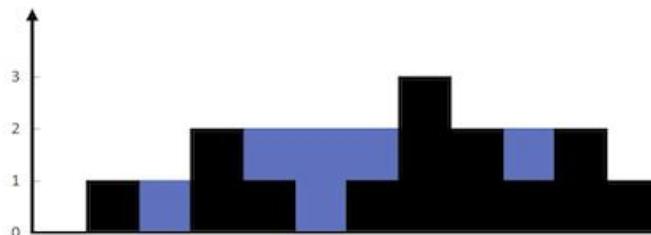
```
class Solution:
    def maxArea(self, height: [int]) -> int:
        MaxArea, N = 0, len(height)
        left, right = 0, N - 1
        while left < right:
            high = min(height[left], height[right])
            di = right - left # 底是由高决定的
            MaxArea = max(MaxArea, high * di)
            if height[left] == high: # 哪边矮动哪边
                left += 1
            else:
                right -= 1
        return MaxArea
```

接雨水

42. 接雨水

难度 困难 1380 收藏 分享 切换为英文 关注 反馈

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。



上面是由数组 `[0,1,0,2,1,0,1,3,2,1,2,1]` 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。感谢 Marcos 贡献此图。

示例：

```
输入: [0,1,0,2,1,0,1,3,2,1,2,1]
输出: 6
```

双指针问题，`left`，`right` 分别记录当前柱子的高度，根据木桶理论，与左右较矮柱子的高度差即为当前位置能够蓄水的体积，所以从较矮柱子那边开始计算雨水的体积，来回切换，直到左指针大于右指针。

如果用单指针，每次都要计算两侧的 `leftMax` 和 `rightMax`，那么时间复杂度将是 $O(n^2)$ 。

```
class Solution:
    def trap(self, height: [int]) -> int:
        N = len(height)
        if N == 0:
            return 0
        leftMax, rightMax = height[0], height[N-1]
        left, right, sum = 1, N - 2, 0 # 从第二个和倒数第二个开始计算，因为第一个和倒数第一个肯定不蓄水
        while left <= right:
            if leftMax < rightMax: # 哪边的最大高度低一点就先计算哪边
                if height[left] < leftMax:
                    sum += leftMax - height[left]
                else:
                    leftMax = height[left] # 左右转换的变量
                left += 1
            else:
                if height[right] < rightMax:
                    sum += rightMax - height[right]
                else:
                    rightMax = height[right]
                right -= 1
        return sum
```

在排序数组中查找元素的第一个位置和最后一个位置

34. 在排序数组中查找元素的第一个和最后一个位置

难度 中等 468 收藏 分享 切换为英文 关注 反馈

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 $O(\log n)$ 级别。

如果数组中不存在目标值，返回 `[-1, -1]`。

示例 1:

```
输入: nums = [5,7,7,8,8,10], target = 8
输出: [3,4]
```

示例 2:

```
输入: nums = [5,7,7,8,8,10], target = 6
输出: [-1,-1]
```

题目要求时间复杂度为 $O(\log N)$ ，很明显要用到二分查找的思想，

`mid = mid // 2 # 左滑到 left`

`If target > nums[mid]:`

`Left = mid + 1 ;`

`Else :`

`Right = mid`

```
class Solution:
    def searchRange(self, nums: [int], target: int) -> [int]:
        N = len(nums)
        if N == 0:
            return [-1, -1]
        left1, left2, right1, right2 = 0, 0, N - 1, N - 1
        while left1 < right1:
            mid = (left1 + right1) // 2 # mid 优先取左边
            if nums[mid] < target:
                left1 = mid + 1 # left 打头+1
            else:
                right1 = mid

        while left2 < right2:
            mid2 = (left2 + right2) // 2 + 1 # mid 优先取右边
            if target < nums[mid2]:
                right2 = mid2 - 1 # right 打头+ 1
            else:
                left2 = mid2

        if nums[left1] == target and nums[left2] == target:
            return [left1, left2]
        else:
            return [-1, -1]
```

奇偶分割数组

905. 按奇偶排序数组

难度 简单 146 收藏 分享 切换为英文 关注 反馈

给定一个非负整数数组 A，返回一个数组，在该数组中，A 的所有偶数元素之后跟着所有奇数元素。

你可以返回满足此条件的任何数组作为答案。

示例：

```
输入：[3,1,2,4]
输出：[2,4,3,1]
输出 [4,2,3,1], [2,4,1,3] 和 [4,2,1,3] 也会被接受。
```

```
class Solution:
    def sortArrayByParity(self, nums):
        start, end = 0, len(nums) - 1
        while start < end:
            while start < end and nums[start] % 2 == 0: # 找到奇数为止
                start += 1
            while start < end and nums[end] % 2 == 1: # 找到偶数为止
                end -= 1
            if start < end:
                nums[start], nums[end] = nums[end], nums[start]
                start += 1
                end -= 1
        return nums
```

字符串、数组、集合

最长公共前缀

14. 最长公共前缀

难度 简单 1200 收藏 分享 切换为英文 关注 反馈

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""。

示例 1：

```
输入: ["flower", "flow", "flight"]
输出: "fl"
```

示例 2：

```
输入: ["dog", "racecar", "car"]
输出: ""
解释: 输入不存在公共前缀。
```

说明：

所有输入只包含小写字母 a-z。

```
class Solution:
    def longestCommonPrefix(self, strs: [str]) -> str:
        if len(strs) == 0:
            return ""
        maxPrefix = strs[0]
        for i in range(1, len(strs)):
            while strs[i].find(maxPrefix) != 0: # 直到找到公共前缀
                maxPrefix = maxPrefix[:-1] # 去掉最后一个元素后，仍然是公共前缀
        return maxPrefix
```

字符串分割（华为）

题目描述

连续输入字符串(输出次数为N,字符串长度小于100)，请按长度为8拆分每个字符串后输出到新的字符串数组，长度不是8整数倍的字符串请在后面补数字0，空字符串不处理。

首先输入一个整数，为要输入的字符串个数。

例如：

输入：2

abc

12345789

输出：abc00000

12345678

90000000

关键：8 的余数单独计算

处理输入输出时一定要用

```
while True:
```

```
    try:
```

```
        except:
```

```
            break
```

```
class Solution():
    def strSplit(self, List):
        for x in List:
            l = len(x)
            y = l % 8
            for i in range(0, l - y, 8):
                print(x[i:i + 8])
            if y != 0:
                print(x[l - y:] + '0' * (8 - y))

if __name__ == '__main__':
    solution = Solution()
    while True:
        try:
            N = int(input())
            List = list()
            for _ in range(N):
                List.append(input())
            solution.strSplit(List)
        except:
            break
```

外观数列

38. 外观数列

难度 简单 479 收藏 分享 切换为英文 关注 反馈

给定一个正整数 n ($1 \leq n \leq 30$)，输出外观数列的第 n 项。

注意：整数序列中的每一项将表示为一个字符串。

「外观数列」是一个整数序列，从数字 1 开始，序列中的每一项都是对前一项的描述。前五项如下：

```
1.      1
2.      11
3.      21
4.      1211
5.      111221
```

第一项是数字 1

描述前一项，这个数是 1 即“一个 1”，记作 11

描述前一项，这个数是 11 即“两个 1”，记作 21

描述前一项，这个数是 21 即“一个 2 一个 1”，记作 1211

描述前一项，这个数是 1211 即“一个 1 一个 2 两个 1”，记作 111221

示例 1：

```
输入: 1
输出: "1"
解释: 这是一个基本样例。
```

示例 2：

```
输入: 4
输出: "1211"
解释: 当 n = 3 时, 序列是 "21", 其中我们有 "2" 和 "1" 两组, "2" 可以读作 "12", 也就是出现频次 = 1 而值 = 2; 类似 "1" 可以读作 "11"。所以答案是 "12" 和 "11" 组合在一起, 也就是 "1211"。
```

```
class Solution(object):
    def countAndSay(self, n):
        char = '1' # 第一项
        for i in range(n - 1): # 第一项不用计算
            tempchar = ''
            tempN = 1 # 计数
            for j in range(len(char)):
                if j + 1 < len(char) and char[j] == char[j + 1]: # 限制j+1不出界
                    tempN += 1
                else:
                    tempchar += str(tempN) + char[j]
                    tempN = 1 # 重新计数
            char = tempchar
        return char
```

动画描述



复杂度分析

- 空间复杂度: $O(1)$
- 时间复杂度: $O(1)$

坐标移动

题目描述

开发一个坐标计算工具，A表示向左移动，D表示向右移动，W表示向上移动，S表示向下移动。从(0,0)点开始移动，从输入字符串里面读取一些坐标，并将最终输入结果输出到输出文件里面。

输入：

合法坐标为A(或者D或者W或者S) + 数字（两位以内）

坐标之间以分隔。

非法坐标点需要进行丢弃。如AA10; A1A; \$%\$; YAD; 等。

下面是一个简单的例子如：

A10;S20;W10;D30;X;A1A;B10A11;;A10;

处理过程：

起点 (0,0)

+ A10 = (-10,0)

+ S20 = (-10,-20)

+ W10 = (-10,-10)

+ D30 = (20,-10)

+ X = 无效

+ A1A = 无效

+ B10A11 = 无效

+ 一个空 不影响

+ A10 = (10,-10)

结果 (10, -10)

注意请处理多组输入输出

输入描述:

一行字符串

输出描述:

最终坐标，以, 分隔

示例1

```
输入 A10;S20;W10;D30;X;A1A;B10A11;;A10;  
输出 10, -10
```

思路：判断字符串是否是数字用 `isdigit()`，判断是否是字母用 `isalpha`，判断是否是数字和字母组成用用 `isalnum`，判断是否是大小写字母，判断是否是空格都有类似的方法。

此外

编程题中最好设置循环判断多个 case 的 while True : try: except: break 结构体。

```
while True:  
    try:  
        List = input().strip().split(";;")  
        x, y = 0, 0  
        for s in List:  
            if len(s) > 3 or len(s) < 2:  
                continue  
            if s[0] not in ['A', 'D', 'W', 'S']:  
                continue  
            if not s[1:].isdigit():  
                continue  
            if s[0] == 'A':  
                x -= int(s[1:])  
            elif s[0] == 'D':  
                x += int(s[1:])  
            elif s[0] == 'W':  
                y += int(s[1:])  
            else:  
                y -= int(s[1:])  
        print(str(x) + ',' + str(y))  
    except:  
        break
```

字符串相加

415. 字符串相加

难度 简单 179 收藏 分享 切换为英文 关注 反馈

给定两个字符串形式的非负整数 `num1` 和 `num2`，计算它们的和。

注意：

1. `num1` 和 `num2` 的长度都小于 5100.
2. `num1` 和 `num2` 都只包含数字 0-9 .
3. `num1` 和 `num2` 都不包含任何前导零。
4. 你不能使用任何內建 BigInteger 库，也不能直接将输入的字符串转换为整数形式。

```
class Solution:  
    def addStrings(self, num1: str, num2: str) -> str:  
        List1, List2, carry, res = list(num1), list(num2), 0, ""  
        while List1 or List2 or carry != 0: # carry不等于0能够将最后的进位放在前面  
            a = int(List1.pop()) if List1 else 0  
            b = int(List2.pop()) if List2 else 0  
            v = (a + b + carry) % 10  
            carry = (a + b + carry) // 10  
            res = str(v) + res  
        return res
```

丑数

263. 丑数

难度 简单 | 130 收藏 分享 切换为英文 | 关注 | 反馈

编写一个程序判断给定的数是否为丑数。

丑数就是只包含质因数 2, 3, 5 的正整数。

示例 1:

```
输入: 6
输出: true
解释: 6 = 2 × 3
```

示例 2:

```
输入: 8
输出: true
解释: 8 = 2 × 2 × 2
```

示例 3:

```
输入: 14
输出: false
解释: 14 不是丑数，因为它包含了另外一个质因数 7。
```

说明：

- 1 是丑数。
- 输入不会超过 32 位有符号整数的范围: [-2³¹, 2³¹ - 1]。

```
class Solution:
    def isUgly(self, n):
        if n <= 0:
            return False
        while n % 2 == 0: # 一直取余，如果等于0，说明还有该数的因子，所以要整除掉它
            n /= 2
        while n % 3 == 0:
            n /= 3
        while n % 5 == 0:
            n /= 5
        return n == 1
```

栈

移除 k 位数字

402. 移掉K位数字

难度 中等 271 收藏 分享 切换为英文 关注 反馈

给定一个以字符串表示的非负整数 num ，移除这个数中的 k 位数字，使得剩下的数字最小。

注意:

- num 的长度小于 10002 且 $\geq k$ 。
- num 不会包含任何前导零。

示例 1：

```
输入: num = "1432219", k = 3
输出: "1219"
解释: 移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219。
```

示例 2：

```
输入: num = "10200", k = 1
输出: "200"
解释: 移掉首位的 1 剩下的数字为 200。注意输出不能有任何前导零。
```

示例 3：

```
输入: num = "10", k = 2
输出: "0"
解释: 从原数字移除所有的数字，剩余为空就是0。
```

数字最小，就要尽量移除字符串前面的较大整数。

遍历数组，查看当前遍历元素的前一个元素，如果当前元素小于前一个元素，则移除前一个元素，直到移除 k 个元素位置。

注意：如果字符串最后一段是升序的，那么可能移除的元素达不到 k ，这个时候需移除后面的元素使移除元素个数达到 k 。

技巧：利用栈来移除和添加元素最佳

```
class Solution:
    def removeKdigits(self, num: str, k: int) -> str:
        N, stack, = len(num), []
        stack.append(num[0]) # 保证栈不为空, 才能移除元素
        remain = N - k # 假如字符串某段升序, 那么可能不会出栈k次, 所以需要采用截取的方法得到结果
        for i in range(1, N):
            while k and stack and stack[-1] > num[i]: # 如果栈尾元素大于下一个元素, 移除一个数
                stack.pop()
                k -= 1
            stack.append(num[i]) # 不论如何这个数都是要加到栈中的
        res = ''.join(stack[:remain]).lstrip('0')
        return res if res else '0' # 删除前导0之后, 加入为空, 说明最小值是0
```

去除重复字母

316. 去除重复字母

难度 困难 190 收藏 分享 切换为英文 关注 反馈

给你一个仅包含小写字母的字符串，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证返回结果的字典序最小（要求不能打乱其他字符的相对位置）。

示例 1:

输入: "bcabc"
输出: "abc"

示例 2:

输入: "cbacdcbc"
输出: "acdb"

注意：该题与 1081 <https://leetcode-cn.com/problems/smallest-subsequence-of-distinct-characters> 相同

```
class Solution:
    def removeDuplicateLetters(self, s: str) -> str:
        N, stack = len(s), []
        import collections
        remain_counter = collections.Counter(s) # 计数每一个字符出现的次数，返回字典
        for x in s:
            if x not in stack: # 如果该元素已经在栈中，则无需操作
                while stack and stack[-1] > x and remain_counter[stack[-1]] >= 1: # 后面还有该元素，可以弹栈
                    stack.pop()
                stack.append(x)
                remain_counter[x] -= 1 # 该元素已被用到
        return ''.join(stack)
```

拼接最大数

321. 拼接最大数

难度 困难

120

收藏

分享

切换为英文

关注

反馈

给定长度分别为 m 和 n 的两个数组，其元素由 0-9 构成，表示两个自然数各位上的数字。现在从这两个数组中选出 k ($k \leq m + n$) 个数字拼接成一个新的数，要求从同一个数组中取出的数字保持其在原数组中的相对顺序。

求满足该条件的最大数。结果返回一个表示该最大数的长度为 k 的数组。

说明: 请尽可能地优化你算法的时间和空间复杂度。

示例 1:

```
输入:  
nums1 = [3, 4, 6, 5]  
nums2 = [9, 1, 2, 5, 8, 3]  
k = 5  
输出:  
[9, 8, 6, 5, 3]
```

示例 2:

```
输入:  
nums1 = [6, 7]  
nums2 = [6, 0, 4]  
k = 5  
输出:  
[6, 7, 6, 0, 4]
```

示例 3:

```
输入:  
nums1 = [3, 9]  
nums2 = [8, 9]  
k = 3  
输出:  
[9, 8, 9]
```

```

class Solution:
    def maxNumber(self, nums1: List[int], nums2: List[int], k: int) -> List[int]:
        M, N = len(nums1), len(nums2)
        maxList = []
        for i in range(k + 1): # A、B都可以为空
            if i <= len(nums1) and k - i <= len(nums2): # 降低时间复杂度
                A, B = self.findMax(nums1, i), self.findMax(nums2, k - i)
                mergeList = self.merge(A, B)
                if len(mergeList) > len(maxList):
                    maxList = mergeList
                elif len(mergeList) == len(maxList):
                    maxList = max(maxList, mergeList)
                    # 列表大小取决于第一个不同的元素,所以长度相等的列表才能比较大小
        return maxList

    def merge(self, A, B):
        res = []
        while A or B:
            bigger = A if A > B else B # 空数组用于小于非空数组
            res.append(bigger[0])
            bigger.pop(0)
        return res

    def findMax(self, nums, k):
        N, stack = len(nums), []
        remain = N - k
        for num in nums:
            while remain and stack and stack[-1] < num: # 从前到后, 弹出remain - k个较小值
                stack.pop()
                remain -= 1
            stack.append(num)
        return stack[:k] # 可能没有移除remain - k个, 所以后面的可以舍去

```

栈排序

最长有效括号

32. 最长有效括号

难度 困难 847 收藏 分享 切换为英文 关注 反馈

给定一个只包含 '(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度。

示例 1:

输入: "()"
输出: 2
解释: 最长有效括号子串为 "()"

示例 2:

输入: ")()()"
输出: 4
解释: 最长有效括号子串为 "()()"

思路：当 stack 为空时，长度置 0，左括号进栈，遇右括号出栈，长度加 2，直到末尾。

```

class Solution:
    def longestValidParentheses(self, s: str) -> int:
        stack, maxLen, N = [], 0, len(s)
        stack.append(-1) # 赋值一个-1的坐标
        for i in range(N):
            if s[i] == '(':
                stack.append(i)
            if s[i] == ')':
                stack.pop()
            if not stack: # 表明-1被弹出了，需要加一个新的-1
                stack.append(i)
            else:
                maxLen = max(maxLen, i - stack[-1])
        return maxLen

```

三元解析式

给定一个表示任意嵌套三元表达式的字符串 expressions,本例将计算表达式的结果。可以假设给定的表达式是有效的,并且只由数字、T、F 组成(T、F 分别表示 True 和 False)。

需要注意的是:

- 1、给定字符串的长度
- 2、每个整数都是个位数
- 3、条件表达式从右到左(跟大多数语言一样)
- 4、条件永远都是 T 或 F,不会是一个数字
- 5、表达式的结果总是对 0-9、T 或 F 求值。

我的思路:从右往左,将数字和 T、F 存入栈中,栈中只存两个数,遇到问号,往左再看一位,T 选 left , F 选 right, 在将选值存入栈中, 知道走到了最左边,输出栈中的内容即可。

```
class Solution:
    def parseTernary(self, expression):
        objects = []
        i = len(expression) - 1
        while i >= 0:
            if expression[i] == '?':
                left, right = objects.pop(-1), objects.pop(-1)
                objects.append(left if expression[i - 1] == 'T' else right)
                i -= 1 # 前进一位
            elif expression[i] != ';':
                objects.append(expression[i])
                i -= 1
        print(objects)
        return objects[0]
```

逆波兰表达式求值

150. 逆波兰表达式求值

难度 中等 154 收藏 分享 切换为英文 关注 反馈

根据 逆波兰表示法，求表达式的值。

有效的运算符包括 +, -, *, /。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

- 整数除法只保留整数部分。
- 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

```
输入: ["2", "1", "+", "3", "*"]
输出: 9
解释: 该算式转化为常见的中缀算术表达式为: ((2 + 1) * 3) = 9
```

示例 2：

```
输入: ["4", "13", "5", "/", "+"]
输出: 6
解释: 该算式转化为常见的中缀算术表达式为: (4 + (13 / 5)) = 6
```

示例 3：

```
输入: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
输出: 22
解释:
该算式转化为常见的中缀算术表达式：
((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

逆波兰表达式：

逆波兰表达式是一种后缀表达式，所谓后缀就是指算符写在后面。

- 平常使用的算式则是一种中缀表达式，如 $(1 + 2) * (3 + 4)$ 。
- 该算式的逆波兰表达式写法为 $(1 2 +) (3 4 +) *$ 。

逆波兰表达式主要有以下两个优点：

- 去掉括号后表达式无歧义，上式即便写成 $1 2 + 3 4 + *$ 也可以依据次序计算出正确结果。
- 适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

```
class Solution:
    def evalRPN(self, tokens):
        stack = []
        for i in tokens:
            if i not in ('+', '-', '*', '/'): # 载字入栈
                stack.append(int(i))
            else:
                op2 = stack.pop() # 先出栈的是第二个操作数
                op1 = stack.pop() # 后出栈的是第一个操作数
                if i == '+': stack.append(op1 + op2)
                elif i == '-': stack.append(op1 - op2)
                elif i == '*': stack.append(op1 * op2)
                else: stack.append(int(op1 / op2)) # 题目规定取整
        return stack[0]
```

有效的括号

20. 有效的括号

难度 简单 山 1643 心 收藏 分享 切换为英文 关注 反馈

给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

输入: "()"
输出: true

示例 2:

输入: "()[]{}"
输出: true

示例 3:

输入: "()"
输出: false

示例 4:

输入: "([)]"
输出: false

示例 5:

输入: "[[]]"
输出: true

```
class Solution(object):
    def isValid(self, s):
        length = len(s)
        stack = []
        for i in range(length):
            if s[i] == '(' or s[i] == '[' or s[i] == '{': # 左括号进栈
                stack.append(s[i])
            else:
                if len(stack) == 0: # 防止stack[-1]溢出
                    return False
                elif s[i] == ')' and stack[-1] != '(' or s[i] == ']' and stack[-1] != '[' or s[i] == '}' and stack[-1] != '{':
                    return False
                else: # 遇到右括号时，弹栈的那个括号必须是对应的左括号
                    stack.pop()
        return True if len(stack) == 0 else False # 弹除只有左括号的情况
```

棒球游戏

682. 棒球比赛

难度 简单 | 130 收藏 分享 切换为英文 | 关注 反馈

你现在是棒球比赛记录员。

给定一个字符串列表，每个字符串可以是以下四种类型之一：

1. 整数（一轮的得分）：直接表示您在本轮中获得的积分数。
2. “+”（一轮的得分）：表示本轮获得的得分是前两轮 有效 回合得分的总和。
3. “D”（一轮的得分）：表示本轮获得的得分是前一轮 有效 回合得分的两倍。
4. “C”（一个操作，这不是一个回合的分数）：表示您获得的最后一个 有效 回合的分数是无效的，应该被移除。

每一轮的操作都是永久性的，可能会对前一轮和后一轮产生影响。

你需要返回你在所有回合中得分的总和。

示例 1:

```
输入: ["5", "2", "C", "D", "+"]
输出: 30
解释:
第1轮：你可以得到5分。总和是：5。
第2轮：你可以得到2分。总和是：7。
操作1：第2轮的数据无效。总和是：5。
第3轮：你可以得到10分（第2轮的数据已被删除）。总数是：15。
第4轮：你可以得到5 + 10 = 15分。总数是：30。
```

示例 2:

```
输入: ["5", "-2", "4", "C", "D", "9", "+", "+"]
输出: 27
解释:
第1轮：你可以得到5分。总和是：5。
第2轮：你可以得到-2分。总数是：3。
第3轮：你可以得到4分。总和是：7。
操作1：第3轮的数据无效。总数是：3。
第4轮：你可以得到-4分（第三轮的数据已被删除）。总和是：-1。
第5轮：你可以得到9分。总数是：8。
第6轮：你可以得到-4 + 9 = 5分。总数是13。
第7轮：你可以得到9 + 5 = 14分。总数是27。
```

```
class Solution:
    def calPoints(self, ops):
        # Time: O(n)
        # Space: O(n)
        history = []
        for op in ops:
            if op == 'C':
                history.pop()
            elif op == 'D':
                history.append(history[-1] * 2)
            elif op == '+':
                history.append(history[-1] + history[-2])
            else:
                history.append(int(op))
        return sum(history)
```

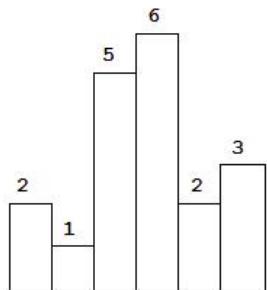
柱状图中最大的矩形

84. 柱状图中最大的矩形

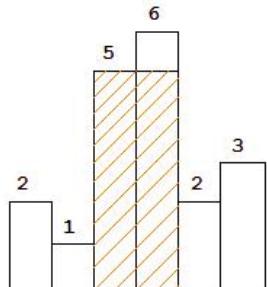
难度 困难 769 收藏 分享 切换为英文 关注 反馈

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 $[2, 1, 5, 6, 2, 3]$ 。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

思路：总是要找尽可能宽和尽可能高的矩形，那么用栈来存储矩形的横坐标，遇到高的就入

栈，因为可能能找到更大的矩形，而遇到矮的就要出栈，此时开始计算矩形的面积，因为这

个面积可能是候选最大矩形。

总之一句话：栈中存坐标，遇到高的入栈，遇到低的出栈，记录矩形大小。

```

class Solution:
    def largestRectangleArea(self, heights):
        if len(heights) == 0:
            return 0
        heights.append(0)      # 末尾添加最低位，使得最后所有栈中的元素都会出栈
        length = len(heights)
        area = 0
        stack = []
        for i in range(length):
            while stack and heights[stack[-1]] > heights[i]:  # 遇到底的就出栈
                hight_index = stack.pop()
                height = heights[hight_index]
                left_index = stack[-1] if stack else -1           # 如果栈为0，那么宽度应该等于弹出位的坐标。
                width = i - left_index - 1  # 多栈不为空时 | 宽度等于当前低位坐标减出栈位坐标
                area = max(area, width * height)
            stack.append(i)      # heights[i]存入栈中时一定是最高的
        return area

```

最大矩形

85. 最大矩形

难度 困难 480 收藏 分享 切换为英文 关注 反馈

给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

示例:

```

输入:
[
    ["1","0","1","0","0"],
    ["1","0","1","1","1"],
    ["1","1","1","1","1"],
    ["1","0","0","1","0"]
]
输出: 6

```

我的思路:把前 n 行当做是一个柱状图，求 n 次最大矩形面积，取最大的那个。

```

class Solution:
    def maximalRectangle(self, matrix):
        if len(matrix) == 0:
            return 0
        if len(matrix[0]) == 0:
            return 0
        rowNum, colNum = len(matrix), len(matrix[0])
        for i in range(rowNum):
            for j in range(colNum):
                matrix[i][j] = int(matrix[i][j])
        newMatrix = [[0] * (colNum + 1) for _ in range(rowNum + 1)] # 行多出第一行，用来上下相加，列多出一列，用来以防序列合并越界，也无法弹出的情况
        for i in range(1, len(newMatrix)):
            for j in range(colNum):
                if matrix[i - 1][j] == 1:# 只有本身是1，才能加上上一行的和
                    newMatrix[i][j] = newMatrix[i - 1][j] + matrix[i - 1][j] # 表示前n行的柱状图高度
        M, N, area = len(newMatrix), len(newMatrix[0]), 0
        for i in range(1, M):
            stack = [] # 栈需要定义成局部变量
            for j in range(N):
                while stack and newMatrix[i][stack[-1]] > newMatrix[i][j]:
                    height_index = stack.pop()
                    height = newMatrix[i][height_index]
                    left_index = stack[-1] if stack else -1
                    width = j - left_index - 1
                    area = max(area, width * height)
                stack.append(j)
        return area

```

栈的压入、弹出序列

剑指 Offer 31. 栈的压入、弹出序列

难度 中等 41 收藏 分享 切换为英文 关注 反馈

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如，序列 {1,2,3,4,5} 是某栈的压栈序列，序列 {4,5,3,2,1} 是该压栈序列对应的一个弹出序列，但 {4,3,5,1,2} 就不可能是该压栈序列的弹出序列。

示例 1：

```

输入：pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
输出：true
解释：我们可以按以下顺序执行：
push(1), push(2), push(3), push(4), pop() -> 4,
push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

```

示例 2：

```

输入：pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
输出：false
解释：1 不能在 2 之前弹出。

```

提示：

1. $0 \leq pushed.length == popped.length \leq 1000$
2. $0 \leq pushed[i], popped[i] < 1000$
3. $pushed$ 是 $popped$ 的排列。

注意：本题与主站 946 题相同：<https://leetcode-cn.com/problems/validate-stack-sequences/>

```

class Solution:
    def validateStackSequences(self, pushed: [int], popped: [int]) -> bool:
        stack, i = [], 0
        for num in pushed:
            stack.append(num) # num 入栈
            while stack and stack[-1] == popped[i]: # 横环判断与出栈
                stack.pop()
                i += 1
        return not stack

```

最小栈

155. 最小栈

难度 简单 583 收藏 分享 切换为英文 关注 反馈

设计一个支持 push , pop , top 操作，并能在常数时间内检索到最小元素的栈。

- push(x) —— 将元素 x 推入栈中。
- pop() —— 删除栈顶的元素。
- top() —— 获取栈顶元素。
- getMin() —— 检索栈中的最小元素。

示例:

输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

输出：
[null,null,null,null,-3,null,0,-2]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); --> 返回 -3.
minStack.pop();
minStack.top(); --> 返回 0.
minStack.getMin(); --> 返回 -2.

提示：

- pop 、 top 和 getMin 操作总是在 非空栈 上调用。

需要额外定义一个栈结构，该栈结构的栈顶即为最小元素

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.minStack = []

    def push(self, x: int) -> None:
        self.stack.append(x)
        tempStack = []
        while self.minStack and self.minStack[-1] < x:# 新加入的元素比栈顶小
            tempStack.append(self.minStack.pop())
        self.minStack.append(x)
        while tempStack:
            self.minStack.append(tempStack.pop())

    def pop(self) -> None:
        top = self.stack.pop()
        tempStack = []
        while self.minStack and self.minStack[-1] != top:
            tempStack.append(self.minStack.pop())
        self.minStack.pop()
        while tempStack:
            self.minStack.append(tempStack.pop())

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.minStack[-1]
```

用两个栈实现队列

剑指 Offer 09. 用两个栈实现队列

难度 简单 86 收藏 分享 切换为英文 关注 反馈

用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead，分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素，deleteHead 操作返回 -1)

示例 1：

```
输入：  
["CQueue","appendTail","deleteHead","deleteHead"]  
[],[3],[],[]  
输出：[null,null,3,-1]
```

示例 2：

```
输入：  
["CQueue","deleteHead","appendTail","appendTail","deleteHead","deleteHead"]  
[],[],[5],[2],[],[]  
输出：[null,-1,null,null,5,2]
```

提示：

- $1 \leq \text{values} \leq 10000$
- 最多会对 appendTail、deleteHead 进行 10000 次调用

其中一个栈只是在移除首元素时用来过渡的

```

class CQueue:
    def __init__(self):
        self.stack1 = []
        self.stack2 = []

    def appendTail(self, value: int) -> None:
        self.stack1.append(value)

    def deleteHead(self) -> int:
        if not self.stack1:
            return -1
        while self.stack1:
            self.stack2.append(self.stack1.pop())
        val = self.stack2.pop()
        while self.stack2:
            self.stack1.append(self.stack2.pop())
        return val

```

HashMap

和为 k 的子数组

560. 和为K的子数组

难度 中等 508 收藏 分享 切换为英文 关注 反馈

给定一个整数数组和一个整数 k ，你需要找到该数组中和为 k 的连续的子数组的个数。

示例 1：

```

输入:nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。

```

说明：

1. 数组的长度为 $[1, 20,000]$ 。
2. 数组中元素的范围是 $[-1000, 1000]$ ，且整数 k 的范围是 $[-1e7, 1e7]$ 。

用前缀和，用空间换时间，但是会超时，所以需要降低时间复杂度。

之前采用的方案是 $\text{pre}[i] - \text{pre}[j] == k$ $i > k$ ，则为一种情况，也就是说还是得遍历，如果是最差的情况，时间复杂度还是会接近 $O(n^2)$ ，所以引入了 hash 表，省去第二遍遍历，让时间复杂度接近 $O(n)$ ，因为 $\text{pre}[j]$ 已知，记录 $\text{pre}[j]$ 的个数，移项 $\text{pre}[j] = \text{pre}[i] - k$ ，然后 count 加上 $\text{pre}[j]$ 的个数即可。

```

from collections import defaultdict

class Solution:
    def subarraySum(self, nums: [int], k: int) -> int:
        count, hash, N, preSum = 0, defaultdict(int), len(nums), [0] * len(nums)
        hash[0] = 1
        for i in range(N):
            if i == 0: # 在循环内分开处理最佳
                preSum[0] = nums[0]
            else:
                preSum[i] += preSum[i - 1] + nums[i] # 这里求前缀和跟计数同时进行是妙笔
            if preSum[i] - k in hash:
                count += hash[preSum[i] - k]
            hash[preSum[i]] += 1 # hash表的更新放到最后
        return count

```

两个数组的交集 I

349. 两个数组的交集

难度 简单 | 210 收藏 分享 切换为英文 | 关注 反馈

给定两个数组，编写一个函数来计算它们的交集。

示例 1：

输入：nums1 = [1,2,2,1], nums2 = [2,2]
输出：[2]

示例 2：

输入：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出：[9,4]

说明：

- 输出结果中的每个元素一定是唯一的。
- 我们可以不考虑输出结果的顺序。

```

class Solution:
    def intersection(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        set1 = set(nums1)
        set2 = set(nums2)
        return list(set2 & set1)

```

两个数组的交集 II

350. 两个数组的交集 II

难度 简单 359 收藏 分享 切换为英文 关注 反馈

给定两个数组，编写一个函数来计算它们的交集。

示例 1：

输入：nums1 = [1,2,2,1], nums2 = [2,2]
输出：[2,2]

示例 2：

输入：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出：[4,9]

说明：

- 输出结果中每个元素出现的次数，应与元素在两个数组中出现次数的最小值一致。
- 我们可以不考虑输出结果的顺序。

进阶：

- 如果给定的数组已经排好序呢？你将如何优化你的算法？
- 如果 `nums1` 的大小比 `nums2` 小很多，哪种方法更优？
- 如果 `nums2` 的元素存储在磁盘上，磁盘内存是有限的，并且你不能一次加载所有的元素到内存中，你该怎么办？

```
from collections import defaultdict

class Solution:
    def intersect(self, nums1: [int], nums2: [int]) -> [int]:
        M, N = len(nums1), len(nums2)
        res = []
        hash = defaultdict(int)
        for i in range(M):
            hash[nums1[i]] += 1 # 统计元素和次数
        for num in nums2:
            if num in hash and hash[num] != 0:
                res.append(num)
                hash[num] -= 1
        return res
```

两个数组的交集（字节）

输出两个数组的交集： 输入：A=[1,4,2,3,5,6,7,3,5]，B=[3,4,2,3,4,6,7]

需要输出：【4,2,3】【6,7】

1,4,2,3,5,6,7,3,5

3,4,2,3,4,6,7

子区间求最大重复数（阿里）

第一行两个整数 n, m 。 $m \leq n \leq 400000$ 。第二行是长度为 n 的数组 a ， $a_i \leq n$ 。
计算 a 有多少个区间（子数组）满足：存在某个数字 v ，使得 v 在数组区间出现次数大于等于 m 。

1 | 输入：
2 | 5 2
3 | 1 2 1 2 5
4 | 输出：
5 | 5
6 | 解释：区间 [1,3][1,4][1,5][2,4][2,5]，要么1出现2次以上，要么2出现超过2次以上。

```
from collections import defaultdict

def find(List, m, n):
    hash = defaultdict(list)
    ans = 0
    for i in range(n):
        hash[List[i]].append(i) # 重复问题基于运用hash表
        if len(hash[List[i]]) >= m:
            ans += n - hash[List[i]][-1]
    print(hash)
    return ans

n, m = map(int, input().strip().split())
while True:
    string = input()
    if string == "":
        break
    List = list(map(int, string.strip().split()))
    print(List)
    print(find(List, m, n))
```

单词规律

290. 单词规律

难度 简单 160 收藏 分享 切换为英文 关注 反馈

给定一种规律 pattern 和一个字符串 str , 判断 str 是否遵循相同的规律。

这里的 遵循 指完全匹配，例如， pattern 里的每个字母和字符串 str 中的每个非空单词之间存在着双向连接的对应规律。

示例1:

```
输入: pattern = "abba", str = "dog cat cat dog"
输出: true
```

示例 2:

```
输入:pattern = "abba", str = "dog cat cat fish"
输出: false
```

示例 3:

```
输入: pattern = "aaaa", str = "dog cat cat dog"
输出: false
```

示例 4:

```
输入: pattern = "abba", str = "dog dog dog dog"
输出: false
```

肯定是利用 hash map , 将 pattern 和 str 一一对应 , 当不对应时返回 False , 但这里最值得注意的一点就是 , 举例来说 pattern = 'abba' , str = 'dog dog dog dog' , 这样的话 , a , b 对应都就是一个了 , 这是不合理的 , 但是确是能通过的 , map{a: dog, b: dog} , 这是一一对应的 , 所以 map.values 中的值也不能相同。

```
class Solution():
    def wordPattern(self, pattern, str):
        map = dict()
        split_arr = str.split(" ")
        if len(split_arr) != len(pattern):
            return False
        for i in range(len(pattern)):
            if pattern[i] not in map: # 不在哈希表中 , 加入新记录
                if split_arr[i] in map.values(): # 不同的pattern不能对应相同的字符
                    return False
                map[pattern[i]] = split_arr[i]
            else: # 在哈希表中 , 判断对应字符是否相等
                if map[pattern[i]] != split_arr[i]:
                    return False
        return True
```

两数之和

1. 两数之和

难度 简单 8488 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那 **两个** 整数，并返回他们的数组下标。
你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

示例:

```
给定 nums = [2, 7, 11, 15], target = 9  
因为 nums[0] + nums[1] = 2 + 7 = 9  
所以返回 [0, 1]
```

暴力求解并不难，如果要达到 $O(n)$ 的时间复杂度的话，那么就只能遍历一次，那么就要用到 hash 表了。这里建立的是 map，分别存储 `nums[i]` 和索引 `i`。

```
class Solution:  
    def twoSum(self, nums: [int], target: int) -> [int]:  
        map = dict() # 声明字典的正确方法  
        if len(nums) < 2:  
            return []  
        for k, v in enumerate(nums):  
            if target - v in map:  
                return [map[target - v], k]  
            else:  
                map[v] = k  
        return []
```

合并表记录

题目描述

数据表记录包含表索引和数值（int范围的整数），请对表索引相同的记录进行合并，即将相同索引的数值进行求和运算，输出按照key值升序进行输出。

输入描述:

先输入键值对的个数
然后输入成对的index和value值，以空格隔开

输出描述:

输出合并后的键值对（多行）

示例1

输入	复制
4 0 1 0 2 1 2 3 4	
输出	复制
0 3 1 2 3 4	

思路：用一个字典存储，当没有该键值时，往里面加，当有该键值时，那么就值加 1 就好，

这里要注意的是，遍历字典需要根据的是字典的 items。

```
if __name__ == '__main__':
    row = int(input())
    result = {}
    for i in range(row):
        List = list(map(int, input().split()))
        if List[0] not in result:
            result[List[0]] = List[1]
        else:
            result[List[0]] += List[1]
    for k, v in result.items():
        print(str(k) + " " + str(v))
```

LRU 缓存

面试题 16.25. LRU缓存

难度 中等 17 收藏 分享 切换为英文 关注 反馈

设计和构建一个“最近最少使用”缓存，该缓存会删除最近最少使用的项目。缓存应该从键映射到值(允许你插入和检索特定键对应的值)，并在初始化时指定最大容量。当缓存被填满时，它应该删除最近最少使用的项目。

它应该支持以下操作：获取数据 `get` 和写入数据 `put`。

获取数据 `get(key)` - 如果密钥 (`key`) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

写入数据 `put(key, value)` - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

示例：

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );  
  
cache.put(1, 1);  
cache.put(2, 2);  
cache.get(1);      // 返回 1  
cache.put(3, 3);  // 该操作会使得密钥 2 作废  
cache.get(2);     // 返回 -1 (未找到)  
cache.put(4, 4);  // 该操作会使得密钥 1 作废  
cache.get(1);     // 返回 -1 (未找到)  
cache.get(3);     // 返回 3  
cache.get(4);     // 返回 4
```

首先确认实现该原理需要的两种数据结构：

1、Map：因为是缓存，就要满足快速读取的功能，所以就要用到 hash 表，而数据一般都是 key-val 形式的，所以用 map，python 的 Map 和 java 的 HashMap 都可。

```
Self.cache = defaultDict(node)  
Self.capacity = maxSize  
Self.size = 0
```

2、双向链表：因为要满足 LRU，这样的话就当缓存满时，移除掉老数据，腾出空间，显然 map 不能满足这样的需求，这个时候就需要使用链表，而为了方便节点的删除和移动，使用双向链表最佳，有三个功能：

A、缓存满时，删除老节点

B、缓存未满，节点不存在缓存，新节点插入到链表头部

C、缓存未满，节点在缓存，将已存在节点移到头部。

```
Class ListNode(object):  
    Def __init__(self, key , val ):
```

```
Self.key = key
Self.val = val
Self.next = None
Self.pre = None
Self.head = ListNode(0, 0)
Self.tail = ListNode(-1, 0)
Self.head.next = self.tail
Self.tail.pre = self.head
```

3、LRU：

```
Class LRU(object):
    Def __init__(self, capacity):
        Self.cache = defaultDict(node)
        Self.capacity = maxSize
        Self.size = 0
        Self.head = ListNode(0, 0)
        Self.tail = ListNode(-1, 0)
        Self.head.next = self.tail
        Self.tail.pre = self.head
```

再来明确 LRU 的两种方法的实现原理：

1、get()：用 map 快速检索数据找到 node，将 node 移动到双向链表头部；

```
Def get(self, key):
    If key not in self.cache:
        Return -1

    node = self.cache[key] # map 和双向链表共用该节点，方便删除
    Self.moveToHead(node)
    Return node.val
```

2、Put()：a、map 能检索到 node，将 node 移动到双向链表头部

b、map 检索不到 node：最大容量是否已满

 否，插入新节点到头部，加入 map

 否，移除链表尾节点，插入新节点到头部，加入 map

```
Def put(self, key, val):
    If key in self.cache:
        Node = self.cache(key)
        Self.moveToHead(Node)

    Else:
        Node = ListNode(key, val)
```

```
If self.size >= self.capacity:  
    Self.removeTail()  
    Self.pushToHead(node)  
Else:  
    Self.pushToHead(node)  
    Self.size += 1
```

3、moveToHead

```
Def moveToHead(self, node):  
    NodePre = node.pre  
    Node.pre.next = node.next  
    node.next.pre = NodePre  
    Node.next = head.next  
    Head.next.pre = Node  
    Head.next = Node  
    Node.pre = head
```

4、pushToHead

```
Def pushToHead(self, node):  
    Node.next = head.next  
    Head.next.pre = node  
    Head.next = node  
    Node.pre = head
```

5、removeTail

```
Def removeTail(self):  
    TailPre = tail.pre.pre  
    TailPre.next = tail  
    Tail.pre = tailPre
```

```
class Node:
    def __init__(self, key=0, val=0): # 这样的话相当于兼容了默认构造函数的作用
        self.key = key
        self.val = val
        self.pre = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.cache = dict()          # map, key是节点的key，val是节点
        self.capacity = capacity     # 最大容量
        self.size = 0                 # 初始容量
        self.head = Node()
        self.tail = Node()
        self.head.next = self.tail   # 双向链表
        self.tail.pre = self.head

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1
        node = self.cache[key]
        self.moveToHead(node)
        return node.val

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            node = self.cache[key]
            self.moveToHead(node)
        else:
            newNode = Node(key, value)
            if self.size <= self.capacity:
                self.putToHead(newNode)
                self.cache[key] = newNode # map和双向链表都要加入
            else:
                self.removeTail()
                self.putToHead(newNode)
                self.cache[key] = newNode
```

```

def removeTail(self):
    last = self.tail.pre
    lastSec = last.pre
    lastSec.next = last.next
    self.tail.pre = lastSec

def putToHead(self, node):
    node.next = self.head.next
    self.head.next.pre = node
    node.pre = self.head
    self.head.next = node

def moveToHead(self, node):
    pre = node.pre
    post = node.next
    pre.next = post
    post.pre = pre      # 先要删除节点
    node.next = self.head.next
    self.head.next.pre = node
    node.pre = self.head
    self.head.next = node # 再插入头部

```

取巧的做法是：利用 LinkedHashMap，设置访问有序

```

class LRUCache extends LinkedHashMap<Integer, Integer>{
    private int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75F, true);
        this.capacity = capacity;
    }

    public int get(int key) {
        return super.getOrDefault(key, -1);
    }

    public void put(int key, int value) {
        super.put(key, value);
    }
}

```

或者利用 Python 的 OrderedDict：

```
class LRUCache(collections.OrderedDict):

    def __init__(self, capacity: int):
        super().__init__()
        self.capacity = capacity

    def get(self, key: int) -> int:
        if key not in self:
            return -1
        self.move_to_end(key)
        return self[key]

    def put(self, key: int, value: int) -> None:
        if key in self:
            self.move_to_end(key)
        self[key] = value
        if len(self) > self.capacity:
            self.popitem(last=False)
```

滑动窗口

待：找到字符串中所有的字母异位词

438. 找到字符串中所有字母异位词

难度 中等 337 收藏 分享 切换为英文 关注 反馈

给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。

字符串只包含小写英文字母，并且字符串 s 和 p 的长度都不超过 20100。

说明：

- 字母异位词指字母相同，但排列不同的字符串。
- 不考虑答案输出的顺序。

示例 1：

输入：
s: "cbaebabacd" p: "abc"

输出：
[0, 6]

解释：
起始索引等于 0 的子串是 "cba"，它是 "abc" 的字母异位词。
起始索引等于 6 的子串是 "bac"，它是 "abc" 的字母异位词。

示例 2：

输入：
s: "abab" p: "ab"

输出：
[0, 1, 2]

解释：
起始索引等于 0 的子串是 "ab"，它是 "ab" 的字母异位词。
起始索引等于 1 的子串是 "ba"，它是 "ab" 的字母异位词。
起始索引等于 2 的子串是 "ab"，它是 "ab" 的字母异位词。

```
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        M, N, res = len(s), len(p), []
        for i in range(M - N + 1):
            if s[i] not in p or s[i + N - 1] not in p:
                continue
            w = s[i : i + N]
            w = sorted(w)
            p = sorted(p)
            if w == p:
                res.append(i)
        return res
```

超时

待：滑动窗口最大值

239. 滑动窗口最大值

难度 困难 476 收藏 分享 切换为英文 关注 反馈

给定一个数组 $nums$ ，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

进阶：

你能在线性时间复杂度内解决此题吗？

示例：

输入: $nums = [1, 3, -1, -3, 5, 3, 6, 7]$, 和 $k = 3$

输出: $[3, 3, 5, 5, 6, 7]$

解释:

滑动窗口的位置	最大值
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

提示：

- $1 \leq nums.length \leq 10^5$
- $-10^4 \leq nums[i] \leq 10^4$
- $1 \leq k \leq nums.length$

```
class Solution:
    def maxSlidingWindow(self, nums: '[int]', k: 'int') -> '[int]':
        # base cases
        n = len(nums)
        if n * k == 0:
            return []
        if k == 1:
            return nums

        def clean_deque(i):
            if deq and deq[0] == i - k:
                deq.popleft()
            while deq and nums[i] > nums[deq[-1]]:
                deq.pop()

        deq = deque()
        max_idx = 0
        for i in range(k):
            clean_deque(i)
            deq.append(i)
            if nums[i] > nums[max_idx]:
                max_idx = i
        output = [nums[max_idx]]

        for i in range(k, n):
            clean_deque(i)
            deq.append(i)
            output.append(nums[deq[0]])
        return output
```

无重复字符的最长子串

3. 无重复字符的最长子串

难度 中等 3920 收藏 分享 切换为英文 关注 反馈

给定一个字符串，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1：

输入： "abcabcbb"
输出： 3
解释： 因为无重复字符的最长子串是 "abc"， 所以其长度为 3。

示例 2：

输入： "bbbbb"
输出： 1
解释： 因为无重复字符的最长子串是 "b"， 所以其长度为 1。

示例 3：

输入： "pwwkew"
输出： 3
解释： 因为无重复字符的最长子串是 "wke"， 所以其长度为 3。
请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。

如果有重复，首部一直出队列

如果没有重复，尾部一直进队列

在这过程中记录最长无重复子串

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        N, maxLen = len(s), 0
        import collections
        queue = collections.deque()
        for i in range(N):
            while s[i] in queue: # 待考察队列中不重复，再添加
                queue.popleft()
            queue.append(s[i])
            maxLen = max(maxLen, len(queue))
        return maxLen
```

堆

数组中第 k 个最大元素

215. 数组中的第K个最大元素

难度 中等 516 收藏 分享 切换为英文 关注 反馈

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1：

输入：[3, 2, 1, 5, 6, 4] 和 $k = 2$
输出：5

示例 2：

输入：[3, 2, 3, 1, 2, 4, 5, 5, 6] 和 $k = 4$
输出：4

说明：

你可以假设 k 总是有效的，且 $1 \leq k \leq$ 数组的长度。

这个问题的话其实很简单，但是如果非得用到数据结构的话，那毫无疑问就是堆，每一种语言几乎都有实现堆的函数，python 里面的话就是 heapq。熟练运用它也是不错的选择。

熟练运用 heapq 的一下几个函数：

Nums = [1, 2, 3, 4]

Heapq.heapfy(Nums) // 对 nums 构建最小堆，如果要构建最大堆，只有将 Nums 里面的数取反。

Heapq.heapPush(Nums, num) // 往最小堆里面加入一个 num 并自动建堆

Heapq.heappop(Nums) // 弹出 nums 中的堆顶元素，即最小元素

Heapq.nlargest(k, Nums) // 弹出列表 nums 中的最大的 k 个元素，注意 Nums 不需要建堆

Heapq.nsmallest(k, nums) // 与上面相反。

```
class Solution:  
    def findKthLargest(self, nums: [int], k: int) -> int:  
        res = heapq.nlargest(k, nums) # 弹出最大的k个元素，并倒序  
        return res[-1]
```

堆正经实现，但是超时

```
class Solution:
    def findKthLargest(self, nums: [int], k: int) -> int:
        N = len(nums)
        for i in range(N - 1, N - k - 1, -1):
            self.heapSort(nums, i)
        nums[0], nums[i] = nums[i], nums[0]
        print(nums)
        return nums[-k]

    def heapSort(self, nums, tail):
        parent = tail // 2 - 1 if tail & 1 == 0 else tail // 2 # 因根节点是0，所以求父节点需分奇偶
        while parent >= 0:
            if 2 * parent + 2 <= tail:
                index = 2 * parent + 1 if nums[2 * parent + 1] >= nums[2 * parent + 2] else 2 * parent + 2
                if nums[index] > nums[parent]:
                    nums[index], nums[parent] = nums[parent], nums[index]
                else:
                    if nums[2 * parent + 1] > nums[parent]:
                        nums[2 * parent + 1], nums[parent] = nums[parent], nums[2 * parent + 1]
            parent -= 1
```

丑数 2

264. 丑数 II

难度 中等 314 收藏 分享 切换为英文 关注 反馈

编写一个程序，找出第 n 个丑数。

丑数就是质因数只包含 2, 3, 5 的正整数。

示例:

```
输入: n = 10
输出: 12
解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。
```

说明:

- 1 是丑数。
- n 不超过1690。

```
import heapq
class Solution:
    def nthUglyNumber(self, n):
        heap = [1] # 最小丑数
        val = 0
        for _ in range(n):
            val = heapq.heappop(heap) # 对heap列表建堆并弹出最小值
            for muti in [2, 3, 5]:
                result = val * muti
                if result not in heap:
                    heapq.heappush(heap, result)
        return val
```

利用堆的性质，每次取出最小值，然后存入堆，再取最小值，循环往复。

注意并不是说光乘以 2 就是就能一直得到最小值，例如 $2 \times 2 \times 2 = 8$ ，而 $2 \times 3 = 6$ ，所以需要分别成 2,3,5.

超级丑数

313. 超级丑数

难度 中等 85 收藏 分享 切换为英文 关注 反馈

编写一段程序来查找第 n 个超级丑数。

超级丑数是指其所有质因数都是长度为 k 的质数列表 `primes` 中的正整数。

示例:

输入: $n = 12$, `primes` = [2, 7, 13, 19]
输出: 32
解释: 给定长度为 4 的质数列表 `primes` = [2, 7, 13, 19]，前 12 个超级丑数序列为：
[1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32]。

说明:

- 1 是任何给定 `primes` 的超级丑数。
- 给定 `primes` 中的数字以升序排列。
- $0 < k \leq 100$, $0 < n \leq 10^6$, $0 < \text{primes}[i] < 1000$ 。
- 第 n 个超级丑数确保在 32 位有符整数范围内。

利用丑数 2 方法会超时

```
import heapq
class Solution:
    def nthSuperUglyNumber(self, n, primes):
        heap = [1]
        val = 0
        for _ in range(n):
            val = heapq.heappop(heap)
            for muti in primes:
                result = muti * val
                if result not in heap:
                    heapq.heappush(heap, result)
        return val
```

利用动态规划也是这个思路，还不如不用。就是拿出最小的依次和质数列表相乘。得到的所

有数再重新由小到大排序。

即第 n 个丑数和质数列表相乘，保证没有遗漏任何数，都将存入堆中，然后建最小堆，得到下一个丑数。

位

灯泡开关 II

672. 灯泡开关 II

难度 中等 48 收藏 分享 切换为英文 关注 反馈

现有一个房间，墙上挂有 n 只已经打开的灯泡和 4 个按钮。在进行了 m 次未知操作后，你需要返回这 n 只灯泡可能有多少种不同的状态。

假设这 n 只灯泡被编号为 $[1, 2, 3 \dots, n]$ ，这 4 个按钮的功能如下：

1. 将所有灯泡的状态反转（即开变为关，关变为开）
2. 将编号为偶数的灯泡的状态反转
3. 将编号为奇数的灯泡的状态反转
4. 将编号为 $3k+1$ 的灯泡的状态反转 ($k = 0, 1, 2, \dots$)

示例 1:

输入: $n = 1, m = 1$.
输出: 2
说明: 状态为: [开], [关]

示例 2:

输入: $n = 2, m = 1$.
输出: 3
说明: 状态为: [开, 关], [关, 开], [关, 关]

示例 3:

输入: $n = 3, m = 1$.
输出: 4
说明: 状态为: [关, 开, 关], [开, 关, 开], [关, 关, 关], [关, 开, 开].

注意： n 和 m 都属于 $[0, 1000]$.

首先要明确这种题目的解题思路：

- 1、一种操作不论执行多少次，最终的状态都和执行 0,1 次结果一样。
- 2、多种操作执行，无关顺序，结果都是一样的。
- 3、取所有操作距离的最小公倍数，在这里也就是 6，也就是说周期为 6， x 和 $x+6$ 的状态是一致的。所以这道题目完全都可以暴力破解枚举出来。

我们看看有哪些操作能改变这六个灯泡的状态：

```
Light 1 = l + a + c + d  
Light 2 = l + a + b  
Light 3 = l + a + c  
Light 4 = l + a + b + d  
Light 5 = l + a + c  
Light 6 = l + a + b
```

可知，其实只需知晓前三个灯的状态就能推断所有灯的状态，所以 $n = \min(n, 3)$

三个灯的状态最大可以获得八种状态，但是不是每一种状态都能获得。

因为四种操作只能取 0/1 次，所以 m 的范围为 [0, 4]，但只要取三种操作，就能得出所有状态了。

当 $n \geq 3$ 时：我们将四种操作这样定义：(0,0,0), (0,1,0), (1,0,1), (1,0,0)

```
m = 0: return 1  
m = 1: return 4  
m = 2: return 7 // 两两异或 6 种，自身异或 1 种  
m = 3: return 8 // 三三异或，能取全八种状态。
```

当 $n == 2$ 时：(0,0),(1,0),(0,1)

```
m = 0 时 : return 1  
m = 1 时 : return 3  
m = 2 时 : return 4  
m = 3 时 : return 4
```

当 $n == 1$ 时 : (0),(1)

```
m = 0 时 : return 1  
m >= 1 时: return 2
```

```
class Solution:  
    def flipLights(self, n: int, m: int) -> int:  
        n = min(n, 3)  
        if m == 0: return 1  
        if m == 1: return [2,3,4][n - 1]  
        if m == 2: return [2,4,7][n-1]  
        else: return [2,4,8][n - 1]
```

颠倒二进制位

190. 颠倒二进制位

难度 简单 收藏 分享 切换为英文 关注 反馈

颠倒给定的 32 位无符号整数的二进制位。

示例 1：

输入： 00000010100101000001111010011100

输出：00111001011110000010100101000000

解题：输入的二进制串 00000010100101000001111010011100 表示无符号整数 43261596。

因此返回 964176192，其二进制表示形式为 00111001011110000010100101000000。

示例 2：

输出：1011111111111111111111111111

因此返回 3221225471，其二进制表示形式为 10111111111111111111111111111111。

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
 - 在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的 [示例 2](#) 中，输入表示有符号整数 `-3`，输出表示有符号整数 `-1073741825`。

进阶

如果多次调用这个函数，你将如何优化你的算法？

1、用 n&1 获得最后一位

2、用 $n \gg 1$ 舍去最后一位

3、用 ret 接受最后一位的左移位，左移位数从 31 递减

```
class Solution:
    def reverseBits2(self, n):
        ret, power = 0, 31
        while n:
            ret += (n & 1) << power
            n = n >> 1
            power -= 1
        return ret
```

类似于取余获得个位数，整除舍去个位数。

有序数组中的单一元素

解法一：

```
class Solution(object):
    def singleNonDuplicate(self, nums):  # 利用位异或的方式
        N = len(nums)
        result = nums[0]
        for i in range(1, N):
            result ^= nums[i]
        return result
```

解法二：

- 算法流程：

- $mid = (left + right) // 2$
- 当 mid 为偶数时， $nums[mid]$ 前面有 **偶数个** 元素，此时分为两种情况：
(1) $nums[mid] == nums[mid+1]$ ，例如：数组 [1, 1, 2, 2, 3, 3, 4, 5, 5], $mid = 4$, $nums[4] == nums[5] == 3$ ；说明 $nums[mid]$ 前面的元素都出现了两次，所以要在 $nums[mid]$ 后面找单一元素，于是更新区间： $left = mid + 1$ ；
(2) $nums[mid] != nums[mid+1]$ ，例如：数组 [1, 1, 2, 3, 3, 4, 4, 5, 5], $mid = 4$, ($nums[4] == 3$) != ($nums[5] == 4$)；说明 $nums[mid]$ 前面的元素中存在单一元素，于是更新区间： $right = mid - 1$ ；
- 当 mid 为奇数时， $nums[mid]$ 前面有 **奇数个** 元素，此时同样也分为两种情况：
(1) $nums[mid] == nums[mid+1]$ ，例如：数组 [1, 1, 2, 3, 3, 4, 4, 5, 5], $mid = 3$, $nums[3] == nums[4] == 3$ ；说明 $nums[mid]$ 前面的元素中存在单一元素，于是更新区间： $right = mid - 1$ ；
(2) $nums[mid] != nums[mid+1]$ ，例如：数组 [1, 1, 2, 2, 3, 3, 4, 5, 5], $mid = 3$, ($nums[3] == 2$) != ($nums[4] == 3$)；说明单一元素出现在 $nums[mid]$ 的后面，于是更新区间： $left = mid + 1$ 。
- 注意边界条件，当 $mid + 1$ 超出边界时，说明单一元素是数组的最后一个元素。

- 复杂度分析：

- 时间复杂度 $O(\log n)$ ：二分查找的时间复杂度为 $O(\log n)$ 。
- 空间复杂度 $O(1)$ ：没有使用额外的空间。

```
class Solution:
    def singleNonDuplicate(self, nums: List[int]) -> int:
        if len(nums) == 1:
            return nums[0]
        left = 0
        right = len(nums)-1
        while left <= right:
            mid = (left + right) // 2
            if mid % 2 == 0 and mid + 1 < len(nums): # mid是偶数(nums[mid]前面有偶数个元素)
                if nums[mid] == nums[mid+1]: # mid前面没有单一元素
                    left = mid + 1
                else: # mid前面有单一元素
                    right = mid - 1
            elif mid % 2 != 0 and mid + 1 < len(nums): # mid是奇数(nums[mid]前面有奇数个元素)
                if nums[mid] == nums[mid+1]: # mid前面有单一元素
                    right = mid - 1
                else: # mid前面没有单一元素
                    left = mid + 1
            else:
                return nums[mid]
        return nums[left]
```

位的基础概念

02

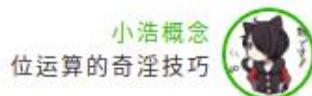
PART

位运算的奇淫技巧



上面的内容相对比较常规，但是一般面试我们遇到的，都不是常规内容。所以下面这些，是必须掌握的。

下面的这八个技巧，基本cover了位运算90%的面试题：



- 1、使用 $x \& 1 == 1$ 判断奇偶数。（注意，一些编辑器底层会把用%判断奇偶数的代码，自动优化成位运算）
- 2、不使用第三个数，交换两个数。 $x = x ^ y$, $y = x ^ y$, $x = x ^ y$ 。（早些年喜欢问到，现在如果谁再问，大家会觉得很low）
- 3、两个相同的数异或的结果是 0，一个数和 0 异或的结果是它本身。（对于找数这块，异或往往有一些别样的用处。）
- 4、 $x \& (x - 1)$ ，可以将最右边的 1 设置为 0。（这个技巧可以用来检测 2 的幂，或者检测一个整数二进制中 1 的个数，又或者别人问你一个数变成另一个数其中改变了多少个bit位，统统都是它）
- 5、异或可以被当做无进位加法使用，与操作可以用来获取进位。
- 6、 $i + (-i) = -1$ ， i 取反再与 i 相加，相当于把所有二进制位设为 1，其十进制结果为 -1。
- 7、对于 int32 而言，使用 $n >> 31$ 取得 n 的正负号。并且可以通过 $(n ^ (n >> 31)) - (n >> 31)$ 来得到绝对值。（ n 为正， $n >> 31$ 的所有位等于 0。若 n 为负数， $n >> 31$ 的所有位等于 1，其值等于 -1）

8、使用 $(x \wedge y) \geq 0$ 来判断符号是否相同。（如果两个数都是正数，则二进制的第一位均为0， $x \wedge y = 0$ ；如果两个数都是负数，则二进制的第一位均为1； $x \wedge y = 0$ 如果两个数符号相反，则二进制的第一位相反， $x \wedge y = 1$ 。有0的情况例外， \wedge 相同得0，不同得1）

位运算计算两个数的和

371. 两整数之和

难度 简单 | 292 收藏 | 分享 | 切换为英文 | 关注 | 反馈

不使用运算符 + 和 -，计算两整数 a、b 之和。

示例 1：

```
输入: a = 1, b = 2
输出: 3
```

示例 2：

```
输入: a = -2, b = 3
输出: 1
```

```
class Solution:
    def getSum(self, a: int, b: int) -> int:
        while b != 0:
            temp = a ^ b # 第一次是a、b两数相加，后面都是和进位值相加，temp存的是相加后的结果
            b = (a & b) << 1 # b为进位值，没有进位了就跳出循环
            a = temp
        return a
```

上述算法只适用于两个正数相加

```
class Solution(object):
    def getSum(self, a, b):
        # 2^32
        MASK = 0x100000000
        # 整型最大值
        MAX_INT = 0x7FFFFFFF
        MIN_INT = 0x80000000
        while b != 0:
            # 计算进位
            carry = (a & b) << 1
            # 取余范围限制在 [0, 2^32-1] 范围内
            a = (a ^ b) % MASK
            b = carry % MASK
        return a if a <= MAX_INT else ~((a % MIN_INT) ^ MAX_INT)
```

二进制求和

67. 二进制求和

难度 简单 431 收藏 分享 切换为英文 关注 反馈

给你两个二进制字符串，返回它们的和（用二进制表示）。

输入为 非空 字符串且只包含数字 1 和 0。

示例 1：

输入: a = "11", b = "1"
输出: "100"

示例 2：

输入: a = "1010", b = "1011"
输出: "10101"

```
class Solution:
    def addBinary(self, a: str, b: str) -> str:
        carry, sum = 0, ""
        M, N = len(a) - 1, len(b) - 1
        while M >= 0 or N >= 0 or carry != 0:# 这样不为0
            int_a = int(a[M]) if M >= 0 else 0# 处理短的字符串或0
            int_b = int(b[N]) if N >= 0 else 0
            cur = (int_a + int_b + carry) % 2
            carry = (int_a + int_b + carry) // 2
            sum = str(cur) + sum
            M -= 1
            N -= 1
        return sum
```

重新排序得到 2 的幂

869. 重新排序得到 2 的幂

难度 中等 32 收藏 分享 切换为英文 关注 反馈

给定正整数 N ，我们按任何顺序（包括原始顺序）将数字重新排序，注意其前导数字不能为零。

如果我们可以通过上述方式得到 2 的幂，返回 true；否则，返回 false。

示例 1：

输入：1
输出：true

示例 2：

输入：10
输出：false

示例 3：

输入：16
输出：true

示例 4：

输入：24
输出：false

示例 5：

输入：46
输出：true

```
from itertools import permutations

class Solution:
    def reorderedPowerOf2(self, N: int) -> bool:
        strN = str(N)
        strList = list(permutations(strN, len(strN))) # 直接给字符串全排列输出字符串数组
        for s in strList:
            if s[0] == '0':
                continue
            s = ''.join(s)
            i = int(s)
            if i & (i - 1) == 0:
                return True
        return False
```

是否是 2 的幂

我们对两组数求“&”运算：

2	0	0	0	0	1	0
1	0	0	0	0	0	1
2&1	0	0	0	0	0	0
8	0	0	1	0	0	0
7	0	0	0	1	1	1
8&7	0	0	0	0	0	0
4	0	0	0	1	0	0
3	0	0	0	0	1	1
4&3	0	0	0	0	0	0
16	0	1	0	0	0	0
15	0	0	1	1	1	1
16&15	0	0	0	0	0	0

可以看到，对于N为2的幂的数，都有 $N \& (N-1) = 0$ ，所以这就是我们的判断条件。（这个技巧可以记忆下来，在一些别的位运算的题目中也是会用到的）

根据分析，完成代码：

```
1 //go
2 func isPowerOfTwo(n int) bool {
3     return n > 0 && n&(n-1) == 0
4 }
```

汉明距离

461. 汉明距离

难度 简单 收藏 分享 切换为英文 关注 反馈

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y ，计算它们之间的汉明距离。

注意：

$0 \leq x, y < 2^{31}$.

示例：

输入: $x = 1, y = 4$

输出: 2

解释:

1 (0 0 0 1)
4 (0 1 0 0)
↑ ↑

上面的箭头指出了对应二进制位不同的位置。

肯定不能移动指针，而应该移动两个数，通过与运算得出两个数最后一个位置的数是啥，然后异或求不同的位数。

```
class Solution:  
    def hammingDistance(self, x: int, y: int) -> int:  
        count = 0  
        for i in range(32):  
            count += (x & 1) ^ (y & 1)  
            x = x >> 1  
            y = y >> 1  
        return count
```

只出现一次的数字 II

137. 只出现一次的数字 II

难度 中等 391 收藏 分享 切换为英文 关注 反馈

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现了三次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

示例 1：

```
输入: [2,2,3,2]
输出: 3
```

示例 2：

```
输入: [0,1,0,1,0,1,99]
输出: 99
```

解法一：

外层循环是位数，如 64 位，内层循环为所有的数字，这样就能计算每一位有多少个 1，

然后将 1 的个数取 3 的余数，那么剩下的就是那个奇数位的数字的 1，然后化为十进制。

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        res = 0
        for i in range(64):
            count = 0
            for x in nums:
                if x & (1 << i) != 0:
                    count += 1
            if count % 3 == 1:
                res += 1 << i
        return res
```

该接发只适用于都是正数的情况

解法二：

将数组去重后，求和然后 $\times 3$ ，然后减去原数组的和，然后 $\text{// } 2$ ，即得到那个只出现一次的数字。

```
class Solution:  
    def singleNumber(self, nums: List[int]) -> int:  
        return (sum(set(nums)) * 3 - sum(nums)) // 2
```

只出现一次的数字 III

先将所有数异或排除所有出现了两次的数，剩下的是两个出现了一次的数的异或结果，将结果只保留最右边一位 1，这个 1 肯定出现在这两个数中的一个，那么现在就变成了多个数中只存在一位数的数目为 1 的情况，找出来，然后将之前的异或结果与其异或就能得到另一个数。

注意：这里有个记忆点，就是保留最右边的一位的方法是：自身和自身的负数相与。

260. 只出现一次的数字 III

难度 中等 265 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 `nums`，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。

示例：

```
输入: [1,2,1,3,2,5]  
输出: [3,5]
```

注意：

1. 结果输出的顺序并不重要，对于上面的例子，`[5, 3]` 也是正确答案。
2. 你的算法应该具有线性时间复杂度。你能否仅使用常数空间复杂度来实现？

```
class Solution:  
    def singleNumber(self, nums: List[int]) -> List[int]:  
        bitmap = 0 # 0和谁异或都会是本身，所以初始化为0  
        for num in nums:  
            bitmap ^= num  
  
        rightMost = bitmap & (-bitmap)  
  
        x = 0 # 存其中一个数  
        for num in nums:  
            if rightMost & num != 0: # 这样就排除了另外一个数  
                x ^= num  
  
        return [x, x ^ bitmap]
```

排序

1. 冒泡排序

```
class Solution():
    def maopaoSort(self, List):
        size = len(List)
        for j in range(size):
            for i in range(1, size - j):
                if List[i] < List[i - 1]:
                    List[i], List[i - 1] = List[i - 1], List[i]
        return List
```

2. 快速排序

```
class Solution():
    def quickSort(self, start, end, List):
        left, right = start, end
        if left >= right:
            return
        pivot = List[start]
        while left < right:
            while left < right and List[right] > pivot:
                right -= 1
            while left < right and List[left] <= pivot:
                left += 1
            if left < right:
                List[left], List[right] = List[right], List[left]
        List[left], List[start] = List[start], List[left]
        self.quickSort(start, right - 1, List)
        self.quickSort(left + 1, end, List)
```

注意事项：

- 1、快排是一个 dfs 无返回值的原地排序，所以需要 left、right 指针和原数组作为参数。
- 2、应该从右指针开始，因为这样就算这是一个原本就已经排好序的数组，那么当右指针扫描到头部时，哨兵也只会和自己交换。
- 3、应该另起一个 left、right 指针取代参数中的 start、end 指针，这样是方便 dfs。

3. 选择排序

```
class Solution():
    def selectSort(self, List):
        size = len(List)
        for i in range(size):
            for j in range(i + 1, size, 1):
                if List[i] > List[j]:
                    List[i], List[j] = List[j], List[i]
```

选择排序就是从后面的数中选择最小的数，那么就应该第一层循环遍历每个数的位置，第二层循环遍历的是该位置后面的位置。

4. 堆排序（选择排序的高级实现）

```
class Solution:
    def heapSort(self, List):
        size = len(List)
        for i in range(size - 1, -1, -1):
            self.buildHeap(List, i) # 对前i个数建堆
            List[0], List[i] = List[i], List[0]
        return List

    def buildHeap(self, List, i):
        parent = (i - 1) // 2
        for j in range(parent, -1, -1):
            if j * 2 + 2 <= i: # 判断是否有右节点
                maxValueIndex = (j * 2 + 2) if List[j * 2 + 2] >= List[j * 2 + 1] else (j * 2 + 1)
                if List[maxValueIndex] > List[j]:
                    List[j], List[maxValueIndex] = List[maxValueIndex], List[j]
                else:
                    continue
            else:
                if List[j * 2 + 1] > List[j]:
                    List[j], List[j * 2 + 1] = List[j * 2 + 1], List[j]
                else:
                    continue
```

堆排序作为选择排序的高级实现：

那么就是每次从堆中选择最大的元素，就能实现升序，反之，将实现降序。

注意一下几点：

1、堆排序是从后往前的，所以循环应该从后往前，因为堆排序也是交换排序，建堆后，

第一个元素为最大元素，那么正好与当前索引元素进行交换。

2、堆排序首先应该找的是最后一个元素的父元素。然后将父元素的值替换成三个元素

中最大的值。当然你要判断是否每个父元素都有左右元素。

3、堆排的话是从最底层的父元素开始到顶点的循环，所以建堆时，初始步骤就是找父元素，然后递减到 0 即可。

5. 插入排序

```
class Solution():
    def charuSort(self, List):
        size = len(List)
        for i in range(1, size):
            for j in range(i - 1, -1, -1):
                if List[j] > List[j + 1]:
                    List[j], List[j + 1] = List[j + 1], List[j]
                else:
                    break
```

插入排序就是将当前位置插入前面排好序的数组中，所以第一层循环应该是从 1 开始的，因为默认第一个元素是已排好序的，接着第二层循环，应该从 $i - 1$ 开始比较，如果该元素小于，那么交换二者，继续往前比较，知道大于，然后跳出第二层循环，所以本质还是交换。

注意：这里进行交换时有一个小技巧就是：

```
For j in range(i - 1, -1, -1):
    If List[j] > List[j + 1]:
        交换二者
```

这样的话，就很完美的进行了插入排序了，记住插入排序的实现原理是，不断交换，而不是真正的插入。

6. 希尔排序（插入排序的高级实现）

就是循环以 $\text{length}/2$ 的步长进行排序，等到步长为 1 时，其实就相当于变成了直接插入排序，但是这个时候需要移动的元素，就很少了，也就是希尔排序改善了直接插入排序。

六、希尔排序

是直接插入排序的一种优化与改进，直接插入排序时，当最小或最大的元素在最后一位时，插入时会移动所有元素，要是该数组长度很长，那么其开销就非常大了，因此出现了希尔排序。

希尔排序整体思想与插入排序一样，只是提出了一种增量或称为步长的概念，不再是直接插入排序那种从头遍历插入的方式，而是以增量跳着插入，一次增量内循环结束后，则增量减半，直到增量小于1，排序结束。直接插入排序的增量可以看作是1，可以以此来更好的理解希尔排序。

以代码中的数组[10, 1, 4, 22, 56, 5, 15, 4]为例，假设其增量为4（一般为数组长度的一半），此时该原始数组被增量切分成许多个新的数组，即[10, 56], [1, 5] ...等等，在一个增量内对该所有“新数组”进行排序，不符合排序规则就交换位置。

这里假设inc代表增量，**希尔排序整体步骤**如下：

- 1、从第inc个元素开始循环遍历其后所有的元素，此时该元素即为待插入元素，
- 2、在已经排序的元素序列中由后往前扫描
- 3、如果该元素（指已排序中的）大于新元素，则该元素往后移
- 4、重复步骤3，直到小于或等于新元素
- 5、将新元素插入到指定位置
- 6、重复步骤2

```
class Solution():
    def shellSorted(self, List):
        size = len(List)
        length = size // 2
        while length != 0:
            for i in range(length, size, length):
                for j in range(i - length, -1, -length):
                    if List[j] > List[j + length]:
                        List[j], List[j + length] = List[j + length], List[j]
            length //= 2
```

7. 归并排序

```
class Solution():
    def mergeSort(self, left, right, List): # 归并排序
        if left >= right: # 最终排序还是剩一个元素时，排序完成，然后利用merge进行排序
            return List[left:right + 1]
        mid = (left + right) // 2
        leftList = self.mergeSort(left, mid, List)
        rightList = self.mergeSort(mid + 1, right, List)
        return self.merge(leftList, rightList)

    def merge(self, leftList, rightList):
        resultList = []
        while leftList and rightList:
            if leftList[0] < rightList[0]:
                resultList.append(leftList.pop(0))
            else:
                resultList.append(rightList.pop(0))
        resultList += leftList
        resultList += rightList
        return resultList
```

归并排序的本质还是 dfs，这个千万记得，与快速排序一样，都需要三个参数，left，right，List。出口是 Left == right。不过相比于快排，归并排序是一个有返回值的非原地排序。

8. 非比较排序

八、非比较排序（分配式排序）

前七种排序算法均为比较排序，即在最终的排序结果里，元素之间的序列依赖于它们之间的比较。几乎每一个元素都需要和其它元素进行比较，才能确定自己的位置。

而非比较排序则通过确定每个元素前面应该有多少个元素进行排序，然后直接在其后插入待排序元素即可。比如针对 $\text{arr}[i]$ ，计算 $\text{arr}[i]$ 之前有多少个元素，则唯一确定了 $\text{arr}[i]$ 所在的位置。例如有五个数小于 $\text{arr}[i]$ ，则 $\text{arr}[i]$ 一定在数组的第六个元素上。

由上面分析可得，非比较排序需要占用大量的空间，因此对待排序数据规模和数据分布有一定要求。而比较排序则适用于一切需要排序的数组。

常见的非比较排序有：计数排序、桶排序、基数排序（内部实现依赖桶排序），本文不对非比较排序算法进行解，因为已经受不了了，这么多排序算法乱七八糟的排排排... 日后得空了再进行总结...

8.1、计数排序

<https://www.jianshu.com/p/86c2375246d7>

9.各种排序算法的时间复杂度

九、各种排序算法的时间复杂度

ps：稳定性是指在待排序数组中，存在多个相同元素，若经过排序后，其相对次序未发生变化，即假设 $a[i]=a[j]$ ，且*i*在*j*前面，那么经过排序后，如果*i*还在*j*前面，则说明该排序是稳定的；反之称为不稳定排序。

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况		
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.5})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	不稳定
归并排序		$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中， r 代表关键字的基数， d 代表长度， n 代表关键字的个数
https://blog.csdn.net/m0_38001814

二叉树

恢复二叉搜索树

99. 恢复二叉搜索树

难度 困难 254 收藏 分享 切换为英文 关注 反馈

二叉搜索树中的两个节点被错误地交换。

请在不改变其结构的情况下，恢复这棵树。

示例 1：

输入： [1, 3, null, null, 2]

```
1
/
3
\
2
```

输出： [3, 1, null, null, 2]

```
3
/
1
\
2
```

示例 2：

输入： [3, 1, 4, null, null, 2]

```
3
/ \
1   4
  /
2
```

输出： [2, 1, 4, null, null, 3]

```
2
/ \
1   4
  /
3
```

进阶：

- 使用 $O(n)$ 空间复杂度的解法很容易实现。
- 你能想出一个只使用常数空间的解决方案吗？

既然只是两个元素，那么可以定义两个全局指针，找到这两个元素，然后这两个节点的值进行相互交换即可。

那么怎么找到这两个节点呢？

诀窍就是中序遍历时：（因为你是从小到大遍历过来的）

遇到第一个当前节点的值小于了上一个节点的值，那么上一个节点的值，就是交换的两个元素中的较大值。

遇到第二个当前节点的值小于上一个节点的值，那么当前节点值，就是两个元素中的较小值。

```
class Solution:
    def recoverTree(self, root: TreeNode) -> None:
        x = y = pre = None # 两个交换的节点指针和前一个元素的指针
        stack = []
        while stack or root:
            if root:
                stack.append(root)
                root = root.left
            else:
                pos = stack.pop()
                if pre != None and pos.val < pre.val:
                    y = pos
                    if x == None:
                        x = pre
                pre = pos
                root = pos.right
        x.val, y.val = y.val, x.val
```

记得非递归中序遍历时，判断的是当前节点是否为空，而不是当前节点的左节点是否为空。

注意这里的巧妙点：

定义三个全局指针：x 指向较大值节点，y 指向较小值节点，pre 指向前一个节点

一定会先碰到较大值节点，所以较小值节点的指针一定会被覆盖写，这是只需将较大值节点的指针设置为不可重复写就可。

删除给定值的叶子节点

1325. 删除给定值的叶子节点

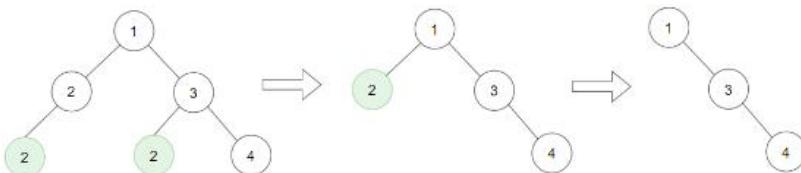
难度 中等 击 27 收藏 分享 切换为英文 关注 反馈

给你一棵以 `root` 为根的二叉树和一个整数 `target`，请你删除所有值为 `target` 的叶子节点。

注意，一旦删除值为 `target` 的叶子节点，它的父节点就可能变成叶子节点；如果新叶子节点的值恰好也是 `target`，那么这个节点也应该被删除。

也就是说，你需要重复此过程直到不能继续删除。

示例 1：



输入：`root = [1,2,3,2,null,2,4]`, `target = 2`

输出：`[1,null,3,null,4]`

解释：

上面左边的图中，绿色节点为叶子节点，且它们的值与 `target` 相同（同为 2），它们会被删除，得到中间的图。

有一个新的节点变成了叶子节点且它的值与 `target` 相同，所以将再次进行删除，从而得到最右边的图。

后序遍历的运用，记得这种涉及到剪枝的问题，都会返回被剪枝好了的树

```
class Solution:
    def removeLeafNodes(self, root: TreeNode, target: int) -> TreeNode:
        if root == None:
            return
        root.left = self.removeLeafNodes(root.left, target)
        root.right = self.removeLeafNodes(root.right, target)
        if not root.left and not root.right and root.val == target:
            return None
        return root # 返回裁剪好的树
```

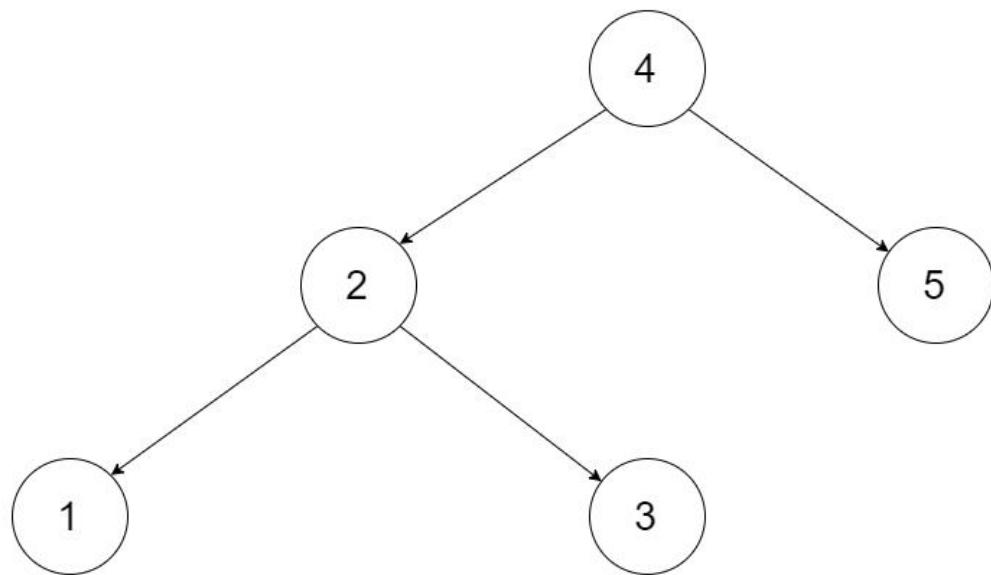
二叉搜索树和双向链表

剑指 Offer 36. 二叉搜索树与双向链表

难度 中等 79 收藏 分享 切换为英文 关注 反馈

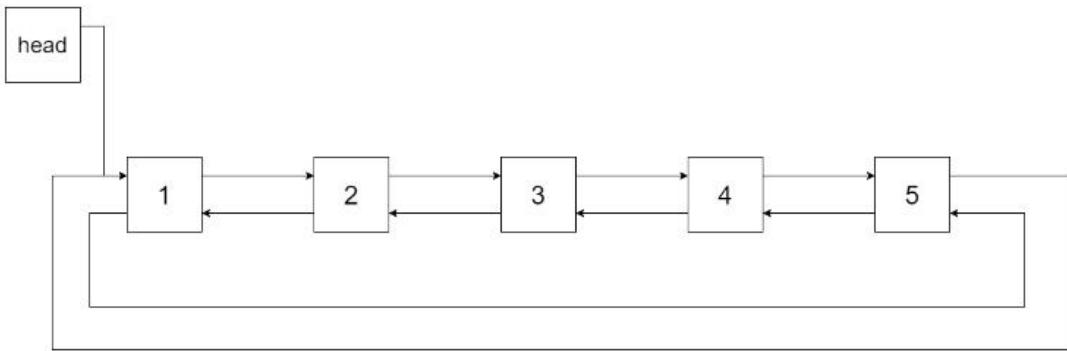
输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。

为了让您更好地理解问题，以下面的二叉搜索树为例：



我们希望将这个二叉搜索树转化为双向循环链表。链表中的每个节点都有一个前驱和后继指针。对于双向循环链表，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

下图展示了上面的二叉搜索树转化成的链表。“head”表示指向链表中有最小元素的节点。



特别地，我们希望可以就地完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中的第一个节点的指针。

注意：本题与主站 426 题相同：<https://leetcode-cn.com/problems/convert-binary-search-tree-to-sorted-doubly-linked-list/>

注意：此题对比原题有改动。

```
class Solution:
    head, tail, pre = None, None, None
    def treeToDoublyList(self, root: 'Node') -> 'Node':
        if root == None:
            return None
        self.inorder(root)
        self.head.left = self.tail
        self.tail.right = self.head
        return self.head

    def inorder(self, root):
        if root == None:
            return
        self.inorder(root.left)
        if self.pre == None: # 无前驱节点，表明为第一个元素
            self.head = root
        else:
            self.pre.right = root
        root.left = self.pre
        self.pre = root
        self.tail = root
        self.inorder(root.right)
```

1、头结点就是前驱节点为空的那个节点，尾节点就是最后遍历的那个节点，所以尾指针要和前驱指针一起随着中序遍历变化而变化。

2、最后 head 和 tail 要连起来。

二叉树的完全性校验

958. 二叉树的完全性检验

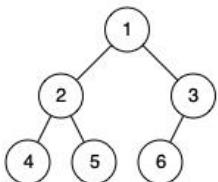
难度 中等 山 63 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，确定它是否是一个完全二叉树。

百度百科中对完全二叉树的定义如下：

若设二叉树的深度为 h ，除第 h 层外，其它各层 ($1 \sim h-1$) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。（注：第 h 层可能包含 $1 \sim 2^h$ 个节点。）

示例 1：

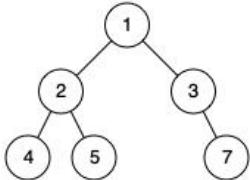


输入：[1, 2, 3, 4, 5, 6]

输出：true

解释：最后一层前的每一层都是满的（即，结点值为 {1} 和 {2, 3} 的两层），且最后一层中的所有结点 ({4, 5, 6}) 都尽可能地向左。

示例 2：



输入：[1, 2, 3, 4, 5, null, 7]

输出：false

解释：值为 7 的结点没有尽可能靠向左侧。

```

class Solution:
    def isCompleteTree(self, root: TreeNode) -> bool:
        import collections
        queue = collections.deque()
        queue.append(root)
        flag = False # 是否出现过null节点
        while queue:
            level = []
            for _ in range(len(queue)):
                pos = queue.popleft()
                if flag and pos != None: # 前面出现空节点，但是弹出元素确是非空节点
                    return False
                if pos != None: # 不为空就有左节点和右节点，只不过可能为空而已
                    level.append(pos.left)
                    level.append(pos.right)
                else:
                    flag = True
            queue.extend(level)
        return True

```

1、层次遍历判断的是 pos 的左右节点不为空，则入队列，而这里是当 pos 不为空时，将左右子节点入栈，那么这样的话，层次遍历时，为空的节点也会存入栈中，毫无疑问，这是一种另外方式的层次遍历，需要注意的是，假设 pos 为空，那么继续记录 flag，再从 queue 中弹出下一个元素。

2、完全二叉树只允许空节点出现在遍历序列的末尾，不允许出现在中间。

求完全二叉树的点数

222. 完全二叉树的节点个数

难度 中等 198 收藏 分享 切换为英文 关注 反馈

给出一个完全二叉树，求出该树的节点个数。

说明：

完全二叉树的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层，则该层包含 $1 \sim 2^h$ 个节点。

示例：

输入：

```

1
/
2  3
/ \
4  5 6

```

输出：6

再来回顾一下满二叉的节点个数怎么计算，如果满二叉树的层数为 h ，则总节点数为： $2^h - 1$ 。

那么我们来对root节点的左右子树进行高度统计，分别记为left和right，有以下两种结果：

1. $\text{left} == \text{right}$ 。这说明，左子树一定是满二叉树，因为节点已经填充到右子树了，左子树必定已经填满了。所以左子树的节点总数我们可以直接得到，是 $2^{\text{left}} - 1$ ，加上当前这个root节点，则正好是 2^{left} 。再对右子树进行递归统计。
2. $\text{left} != \text{right}$ 。说明此时最后一层不满，但倒数第二层已经满了，可以直接得到右子树的节点个数。同理，右子树节点+root节点，总数为 2^{right} 。再对左子树进行递归查找。

```
class Solution:  
    def countNodes(self, root: TreeNode) -> int:  
        if root == None:  
            return 0  
        left = self.countLevel(root.left)  
        right = self.countLevel(root.right)  
        if left == right:# 左子树是满树  
            return self.countNodes(root.right) + (1 << left)# 根节点和-1取消了  
        else:# 右子树是满树  
            return self.countNodes(root.left) + (1 << right)  
  
    def countLevel(self, root):# 求完全二叉树的高度  
        level = 0  
        while root:  
            level += 1  
            root = root.left  
        return level
```

所以求左右子树的高度是关键：

- 1、高度相同，表明左子树为满树
- 2、高度不同，表明右子树为满树
- 3、满树的节点计算公式为 $1 << h - 1$

二叉搜索树中第 k 小的元素

230. 二叉搜索树中第K小的元素

难度 中等 241 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树，编写一个函数 `kthSmallest` 来查找其中第 `k` 个最小的元素。

说明：

你可以假设 `k` 总是有效的， $1 \leq k \leq$ 二叉搜索树元素个数。

示例 1：

```
输入: root = [3,1,4,null,2], k = 1
      3
      / \
     1   4
      \
     2
输出: 1
```

示例 2：

```
输入: root = [5,3,6,2,4,null,null,1], k = 3
      5
      / \
     3   6
      / \
     2   4
      /
     1
输出: 3
```

进阶：

如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第 `k` 小的值，你将如何优化 `kthSmallest` 函数？

```
class Solution:
    def kthSmallest(self, root: TreeNode, k: int) -> int:
        stack, count = [], 0
        while root or stack:
            if root: # 记得这里是root, 不是root.left
                stack.append(root)
                root = root.left
            else:
                pos = stack.pop()
                count += 1
                root = pos.right
                if count == k:
                    return pos.val
        return -1
```

考察非递归遍历中序遍历

二叉树的非递归遍历采取的是栈结构：

其中中序遍历的非递归遍历尤其重要，因为二叉搜索树的中序遍历就是有序的。

牢记这几步：

1、root 非空，就要进栈，栈非空，就能出栈，所以栈为空不能结束循环，需等到 root 也为空。

2、当 root 为空时，就要出栈，表明左边节点为空，需要弹出父节点，然后将 root 赋值为右边节点，继续循环，这个弹出父节点的操作，就是有序序列生成的过程，所以只要计算弹出几次，就能找出第 k 小的元素是几了。

二叉树的最近公共祖先

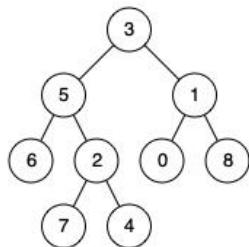
236. 二叉树的最近公共祖先

难度 中等 644 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉树：root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

输出: 3

解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

输出: 5

解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

```

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if root == None or root == p or root == q:
            return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        if left and right:
            return root
        elif not left and right:
            return right
        elif not right and left:
            return left
        else:
            return None

```

后序遍历的极致应用，返回含目标节点的子树，包含的目标节点尽可能多，如果不包含就返回空。

二叉搜索树的最近公共祖先

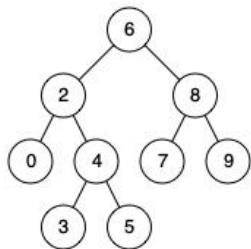
面试题68 - I. 二叉搜索树的最近公共祖先

难度 简单 36 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树，找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉搜索树：root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1：

输入：root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

输出：6

解释：节点 2 和节点 8 的最近公共祖先是 6。

示例 2：

输入：root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

输出：2

解释：节点 2 和节点 4 的最近公共祖先是 2，因为根据定义最近公共祖先节点可以为节点本身。

说明：

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

```

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if (root.val - p.val) * (root.val - q.val) <= 0:
            return root
        if root.val < p.val:
            return self.lowestCommonAncestor(root.right, p, q)
        else:
            return self.lowestCommonAncestor(root.left, p, q)

```

根据二叉搜索树的特性，所以这里可以比较节点值的方法判断，所以应该直接采用前序遍历。

分为三种情况：

- 1、 $p < \text{root} < q$: return root
- 2、 $\text{root} < p$: 往右子树中找
- 3、 $\text{root} > q$: 往左子树中找

平衡二叉树

110. 平衡二叉树

难度 简单 383 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

一个二叉树每个节点的左右两个子树的高度差的绝对值不超过1。

示例 1:

给定二叉树 [3,9,20,null,null,15,7]

```

 3
 / \
 9  20
   / \
  15   7

```

返回 true。

示例 2:

给定二叉树 [1,2,2,3,3,null,null,4,4]

```

 1
 / \
 2   2
 / \
 3   3
 / \
 4   4

```

返回 false。

```

class Solution(object):
    flag = True

    def isBalanced(self, root):
        self.dfs(root)
        return self.flag

    def dfs(self, root): # 返回以root为根节点的深度
        if root == None:
            return 0
        left = self.dfs(root.left)
        right = self.dfs(root.right)
        if abs(left - right) > 1:
            self.flag = False
        return 1 + max(left, right)

```

因为逐层返回 bool 值比较麻烦，所以定义一个全局变量是刚刚好的。

路径总和

112. 路径总和

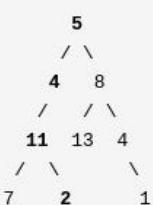
难度 简单 376 收藏 分享 切换为英文 关注 反馈

给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

说明：叶子节点是指没有子节点的节点。

示例：

给定如下二叉树，以及目标和 $\text{sum} = 22$ ，



返回 `true`，因为存在目标和为 22 的根节点到叶子节点的路径 $5 \rightarrow 4 \rightarrow 11 \rightarrow 2$ 。

```

class Solution:
    flag = False

    def hasPathSum(self, root: TreeNode, sum: int) -> bool:
        self.dfs(root, sum)
        return self.flag

    def dfs(self, root, sum):
        if root == None:
            return # 排除根节点为空，因为判断需要用到子节点
        if not root.left and not root.right and sum == root.val: # 前序遍历
            self.flag = True
            return
        self.hasPathSum(root.left, sum - root.val)
        self.hasPathSum(root.right, sum - root.val)

```

路径总和 2

113. 路径总和 II

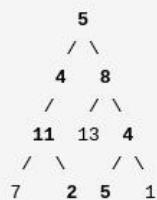
难度 中等 281 收藏 分享 切换为英文 关注 反馈

给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

说明: 叶子节点是指没有子节点的节点。

示例:

给定如下二叉树，以及目标和 $sum = 22$ ，



返回:

```
[  
 [5, 4, 11, 2],  
 [5, 8, 4, 5]  
]
```

```
class Solution:  
    def pathSum(self, root: TreeNode, sum: int) -> [[int]]:  
        if root == None:  
            return []  
        res = []  
        self.dfs(root, sum, [], res)  
        return res  
  
    def dfs(self, root, sum, tmp, res):  
        if root == None: # 给下面的判断做准备  
            return  
        if root.left == None and root.right == None and sum == root.val: # 这样才是叶子节点  
            res.append(tmp[:] + [root.val])  
            return  
        self.dfs(root.left, sum - root.val, tmp + [root.val], res)  
        self.dfs(root.right, sum - root.val, tmp + [root.val], res)
```

路径总和 3

437. 路径总和 III

难度 中等 508 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过1000个节点，且节点数值范围是 [-1000000,1000000] 的整数。

示例：

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

    10
   / \
  5  -3
 / \   \
3  2   11
/ \   \
3  -2   1
```

返回 3。和等于 8 的路径有：

1. 5 -> 3
2. 5 -> 2 -> 1
3. -3 -> 11

```
class Solution:
    ans = 0
    def pathSum(self, root: TreeNode, sum: int) -> int:
        if root == None:
            return 0
        self.dfs(root, sum)      # 前序遍历遍历起点
        self.pathSum(root.left, sum)
        self.pathSum(root.right, sum)
        return self.ans

    def dfs(self, root, sum):
        if root == None:
            return
        if sum == root.val:      # 前序遍历某一个起点开始的路径数
            self.ans += 1
        self.dfs(root.left, sum - root.val)
        self.dfs(root.right, sum - root.val)
```

两次运用先序遍历

二叉搜索子树的最大键值和

1373. 二叉搜索子树的最大键值和

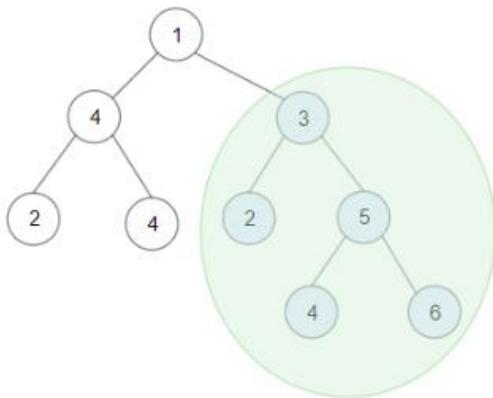
难度 困难 26 收藏 分享 切换为英文 关注 反馈

给你一棵以 `root` 为根的 二叉树 ，请你返回 任意 二叉搜索子树的最大键值和。

二叉搜索树的定义如下：

- 任意节点的左子树中的键值都 小于 此节点的键值。
- 任意节点的右子树中的键值都 大于 此节点的键值。
- 任意节点的左子树和右子树都是二叉搜索树。

示例 1：

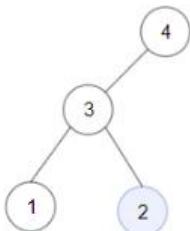


输入：`root = [1,4,3,2,4,2,5,null,null,null,null,null,null,null,4,6]`

输出：20

解释：键值为 3 的子树是和最大的二叉搜索树。

示例 2：



输入：root = [4, 3, null, 1, 2]

输出：2

解释：键值为 2 的单节点子树是和最大的二叉搜索树。

示例 3：

输入：root = [-4, -2, -5]

输出：0

解释：所有节点键值都为负数，和最大的二叉搜索树为空。

示例 4：

输入：root = [2, 1, 3]

输出：6

示例 5：

输入：root = [5, 4, 8, 3, null, 6, 3]

输出：7

阶梯思路：采用的是自底向上，这里的思路很巧，返回三个值，以 root 为根节点的二叉搜索树的路径和，该二叉搜索树的最小值和最大值，这时应熟练处理，当 root 为 None 和当 root 为根节点的二叉树不是二叉搜索树时的情况，将题目条件运用的非常巧妙。

```
class Solution:
    def maxSumBST(self, root: TreeNode) -> int:
        self.maxSum = 0 # 保证了就算全为负数时，也能返回正确结果0
        self.dfs(root)
        return self.maxSum

    def dfs(self, root): # 返回该树的和、最小值、最大值
        if root == None:
            return 0, 5e4, -5e4 # 保证了空节点绝对是一颗二叉搜索树
        leftSum, left_minVal, left_maxVal = self.dfs(root.left)
        rightSum, right_minVal, right_maxVal = self.dfs(root.right) # 自底向上应采取后续遍历
        if left_maxVal < root.val and root.val < right_minVal: # 满足二叉树搜索条件
            sum = root.val + leftSum + rightSum
            self.maxSum = max(self.maxSum, sum)
            return sum, min(root.val, left_minVal), max(root.val, right_maxVal) # 为了空节点加了min和max
        return 0, -5e4, 5e4 # 只要底层有一个不满足，那么上层都不会满足，所以返回一个永远不成立的条件
        | | | | | # 最小值是给右子树用的，最大值是给左子树用的
```

二叉树的最大路径和

124. 二叉树中的最大路径和

难度 困难 562 收藏 分享 切换为英文 关注 反馈

给定一个非空二叉树，返回其最大路径和。

本题中，路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径至少包含一个节点，且不一定经过根节点。

示例 1：

输入： [1, 2, 3]



输出： 6

示例 2：

输入： [-10, 9, 20, null, null, 15, 7]



输出： 42

```
import sys

class Solution(object):
    res = -sys.maxsize
    def maxPathSum(self, root):
        self.dfs(root)
        return self.res

    def dfs(self, root): # 返回以root为起点的单向最大路径和
        if root == None:
            return 0
        left = self.dfs(root.left)
        right = self.dfs(root.right)
        maxSum = root.val + left + right # 经过该点的最大路径 = 左右子树的单向最大路径和 + root.val
        self.res = max(self.res, maxSum) # 记录经过每个点的最长路径，所以需要用到后序遍历
        return max(0, root.val + max(left, right)) # 为负数的话，单向最大路径和为0
```

二叉树的直径

543. 二叉树的直径

难度 简单 385 收藏 分享 切换为英文 关注 反馈

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

示例：

给定二叉树



返回 3, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

```
class Solution(object):
    diam = 0

    def diameterOfBinaryTree(self, root):
        self.dfs(root)
        return self.diam

    def dfs(self, root): # 返回root树的深度
        if root == None:
            return 0
        leftDept = self.dfs(root.left)
        rightDept = self.dfs(root.right)
        self.diam = max(self.diam, leftDept + rightDept)
        return 1 + max(leftDept, rightDept)
```

二叉树中的最长交错路径

1372. 二叉树中的最长交错路径

难度 中等 23 收藏 分享 切换为英文 关注 反馈

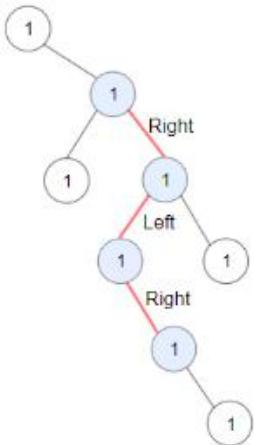
给你一棵以 `root` 为根的二叉树，二叉树中的交错路径定义如下：

- 选择二叉树中 **任意** 节点和一个方向（左或者右）。
- 如果前进方向为右，那么移动到当前节点的的右子节点，否则移动到它的左子节点。
- 改变前进方向：左变右或者右变左。
- 重复第二步和第三步，直到你在树中无法继续移动。

交错路径的长度定义为：访问过的节点数目 - 1（单个节点的路径长度为 0）。

请你返回给定树中最长 **交错路径** 的长度。

示例 1：



输入：`root = [1,null,1,1,1,null,null,1,1,null,1,null,null,null,1,null,1]`

输出：3

解释：蓝色节点为树中最长交错路径（右 -> 左 -> 右）。

题目不难，难得是如何只进行一次深度遍历，解决方法是：

当该往左节点时，那么就不会往右节点了，那么这个时候右节点就可以重新计算最长交错路径，达到一次遍历，两次运用的目的。

因为初始给了一个值并且定义了一个全局值，所以需要用到前序遍历。

第二个参数表示：下一次该往哪一个方向走。

```

class Solution:
    max = 0
    def longestZigZag(self, root: TreeNode) -> int:
        self.dfs(root, 0, 0) # 为0方向向左，为1方向向右
        self.dfs(root, 1, 0)
        return self.max

    def dfs(self, root, dir, length): # 返回以root为起始节点，以dir方向的路径，length为经过边的条数
        if root == None:
            return
        self.max = max(self.max, length)
        if dir == 0:
            self.dfs(root.left, 1, length + 1) # 左节点往右，长度+1
            self.dfs(root.right, 0, 1) # 右节点往左，长度归1
        else:
            self.dfs(root.right, 0, length + 1)
            self.dfs(root.left, 1, 1)

```

二叉树的坡度

563. 二叉树的坡度

难度 简单 69 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，计算整个树的坡度。

一个树的节点的坡度定义即为，该节点左子树的结点之和和右子树结点之和的差的绝对值。空结点的坡度是0。

整个树的坡度就是其所有节点的坡度之和。

示例：

```

输入：
      1
     / \
    2   3
输出：1
解释：
结点 2 的坡度：0
结点 3 的坡度：0
结点 1 的坡度：|2-3| = 1
树的坡度 : 0 + 0 + 1 = 1

```

```

class Solution(object):
    tilt = 0
    def findTilt(self, root):
        self.dfs(root)
        return self.tilt

    def dfs(self, root): # 返回该树的所有节点之和
        if root == None:
            return 0
        left = self.dfs(root.left)
        right = self.dfs(root.right)
        self.tilt += abs(left - right)
        return root.val + left + right

```

合并二叉树

617. 合并二叉树

难度 简单 394 收藏 分享 切换为英文 关注 反馈

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的newValue，否则不为NULL的节点将直接作为新二叉树的节点。

示例 1：

```
输入:
    Tree 1           Tree 2
      1               2
     / \             / \
    3   2           1   3
   /   \
  5     7         4   7

输出:
合并后的树:
      3
     / \
    4   5
   / \   \
  5   4   7
```

注意：合并必须从两个树的根节点开始。

```
class Solution():
    def mergeTrees(self, t1, t2): # 合并t1和t2两棵二叉树
        if t1 == None:
            return t2
        if t2 == None: # 当其中有子树是空树时，返回不是空的那棵树
            return t1
        root = TreeNode(t1.val + t2.val) # 都不为空，合并根节点
        root.left = self.mergeTrees(t1.left, t2.left)
        root.right = self.mergeTrees(t1.right, t2.right)
        return root
```

左子树和左子树合并，右子树和右子树合并，出口是当有一个树为空时，返回另一棵树，当都不为空时，基于两个根节点的和创建新的节点。

也就是说构造了一颗新的树，采取的是前序遍历的思想。

翻转二叉树

226. 翻转二叉树

难度 简单 474 收藏 分享 切换为英文 关注 反馈

翻转一棵二叉树。

示例：

输入：

```
    4
   / \
  2   7
 / \ / \
1  3 6  9
```

输出：

```
    4
   / \
  7   2
 / \ / \
9  6 3  1
```

翻转二叉树就是交换节点的两棵子树，子树里面自己再实现翻转。

```
class Solution:
    def invertTree(self, root):
        self.dfs(root)
        return root

    def dfs(self, node): # 反转以node为根节点的二叉树
        if node == None:
            return
        node.left, node.right = node.right, node.left # 前序遍历
        self.dfs(node.left)
        self.dfs(node.right)
```

根据前序遍历和后序遍历构造二叉树

889. 根据前序和后序遍历构造二叉树

难度 中等 89 收藏 分享 切换为英文 关注 反馈

返回与给定的前序和后序遍历匹配的任何二叉树。

pre 和 post 遍历中的值是不同的正整数。

示例：

```
输入：pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]
输出：[1,2,3,4,5,6,7]
```

提示：

- $1 \leq \text{pre.length} == \text{post.length} \leq 30$
- pre[] 和 post[] 都是 $1, 2, \dots, \text{pre.length}$ 的排列
- 每个输入保证至少有一个答案。如果有多个答案，可以返回其中一个。

```
class Solution:
    def constructFromPrePost(self, pre: [int], post: [int]):
        N = len(pre)
        if N == 0:
            return None
        if N == 1:      # 出口
            return TreeNode(pre[0])
        midPos = 0      # 主要目标是，找到能分割两个序列的中间点，在这里也就是根节点的左子节点
        for i in range(N): # 前序序列中找值，后续序列中找位置
            if post[i] == pre[1]:
                midPos = i + 1
                break
        root = TreeNode(pre[0])
        root.left = self.constructFromPrePost(pre[1 : midPos + 1], post[: midPos])
        root.right = self.constructFromPrePost(pre[midPos + 1 :], post[midPos : -1])
        return root
```

根据中序遍历和后序遍历构造二叉树

106. 从中序与后序遍历序列构造二叉树

难度 中等 231 收藏 分享 切换为英文 关注 反馈

根据一棵树的中序遍历与后序遍历构造二叉树。

注意:

你可以假设树中没有重复的元素。

例如，给出

```
中序遍历 inorder = [9, 3, 15, 20, 7]
后序遍历 postorder = [9, 15, 7, 20, 3]
```

返回如下的二叉树：

```
      3
     / \
    9   20
     /   \
    15   7
```

```
class Solution:
    def buildTree(self, inorder, postorder):
        if not inorder:
            return None
        root = TreeNode(postorder[-1])
        rootPos = inorder.index(postorder[-1])
        root.left = self.buildTree(inorder[:rootPos], postorder[:rootPos])
        root.right = self.buildTree(inorder[rootPos + 1:], postorder[rootPos:-1])
        return root
```

反正根节点不能再包含在 dfs 里面，其余的你自己看着办

二叉树的序列化和反序列化

297. 二叉树的序列化与反序列化

难度 困难 302 收藏 分享 切换为英文 关注 反馈

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例：

你可以将以下二叉树：

```
1
/\ 
2  3
 / \
4  5
```

序列化为 "[1,2,3,null,null,4,5]"

提示：这与 LeetCode 目前使用的方式一致，详情请参阅 [LeetCode 序列化二叉树的格式](#)。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

说明：不要使用类的成员 / 全局 / 静态变量来存储状态，你的序列化和反序列化算法应该是无状态的。

前序遍历

```
class Codec:
    def serialize(self, root): # 返回root树的遍历序列
        if not root:
            return ['null'] # 遇到叶子结点，返回含null的列表即可
        ans = []
        ans.append(str(root.val)) # 前序遍历序列化
        ans += self.serialize(root.left) # 列表相连可以用加号
        ans += self.serialize(root.right)
        return ans

    def deserialize(self, data): # 将data反序列化为树
        ch = data.pop(0) # 根据前序遍历反序列化为树，这里的pop(0)用的很精髓
        if ch == 'null':
            return None
        else:
            root = TreeNode(int(ch)) # 前序遍历构造树
            root.left = self.deserialize(data)
            root.right = self.deserialize(data)
            return root
```

后序遍历（没有通过所有的测试用例，反序列化有点问题）

```

class Codec:
    def serialize(self, root): # 返回root树的遍历序列
        queue, data = [], []
        queue.append(root)
        while queue:
            pos = queue.pop(0)
            if pos != None:
                data.append(pos.val)
                queue.append(pos.left)
                queue.append(pos.right)
            else:
                data.append("None")
        return data

    def deserialize(self, data): # 将data反序列化为树
        return self.bfsBuild(data, 0)

    def bfsBuild(self, data, start):
        N = len(data)
        if start >= N:
            return None
        if data[start] == "None":
            return None
        root = TreeNode(data[start])
        root.left = self.bfsBuild(data, 2 * start + 1)
        root.right = self.bfsBuild(data, 2 * start + 2)
        return root

```

在每个树行中找最大值

515. 在每个树行中找最大值

难度 中等 73 收藏 分享 切换为英文 关注 反馈

您需要在二叉树的每一行中找到最大的值。

示例：

输入：

```

      1
     / \
    3   2
   / \   \
  5   3   9

```

输出： [1, 3, 9]

```
class Solution:
    def largestValues(self, root: TreeNode) -> [int]:
        if root == None:
            return []
        import collections
        queue, result = collections.deque([]), []
        queue.append(root)
        while queue:
            level = []
            for _ in range(len(queue)):
                pos = queue.popleft()
                level.append(pos.val)
                if pos.left:
                    queue.append(pos.left)
                if pos.right:
                    queue.append(pos.right)
            result.append(max(level))
        return result
```

二叉树的右视图

199. 二叉树的右视图

难度 中等 271 收藏 分享 切换为英文 关注 反馈

给定一棵二叉树，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例:

输入: [1,2,3,null,5,null,4]

输出: [1, 3, 4]

解释:



求出每一层的层次遍历，然后返回最右边的值

```

class Solution:
    def rightSideView(self, root: TreeNode) -> [int]:
        if root == None:
            return []
        import collections
        queue, result = collections.deque(), list()
        queue.append(root)
        while queue:
            level = list()
            for _ in range(len(queue)):
                pos = queue.popleft()
                level.append(pos)
                if pos and pos.left != None:
                    queue.append(pos.left)
                if pos and pos.right != None:
                    queue.append(pos.right)
            if len(level) != 0:
                result.append(level[-1].val)
        return result

```

其实这里不用判断 pos 是否为空，因为你 pos 为空压根进不了队列

二叉树的反向层次遍历

107. 二叉树的层次遍历 II

难度 简单 251 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，返回其节点值自底向上的层次遍历。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

例如：

给定二叉树 [3,9,20,null,null,15,7]，

```

3
/
9  20
/   \
15   7

```

返回其自底向上的层次遍历为：

```
[
  [15,7],
  [9,20],
  [3]
]
```

```
class Solution:
    def levelOrderBottom(self, root):
        if root is None:
            return []
        queue = deque([root])
        result = []
        while queue:
            level = []
            size = len(queue)
            for _ in range(size): # 这个逻辑
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            result.append(level)
        return list(reversed(result)) # |||
```

二叉树的锯齿形遍历

103. 二叉树的锯齿形层次遍历

难度 中等 206 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，返回其节点值的锯齿形层次遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：

给定二叉树 [3,9,20,null,null,15,7]，

```
3
 / \
9  20
 / \
15  7
```

返回锯齿形层次遍历如下：

```
[
  [3],
  [20,9],
  [15,7]
]
```

```
class Solution:
    def zigzagLevelOrder(self, root):
        if root is None:
            return []
        import collections
        queue, result = collections.deque(), []
        queue.append(root)
        count = 1
        while queue:
            level = []
            for _ in range(len(queue)):
                pos = queue.popleft()
                level.append(pos.val)
                if pos.left:
                    queue.append(pos.left)
                if pos.right:
                    queue.append(pos.right)
            if count % 2 == 1: # 奇数层顺时针
                result.append(level)
            else: # 偶数层反向遍历
                result.append(level[::-1])
            count += 1
        return result
```

二叉树的垂直遍历

题目描述:

给定一个二叉树，返回其结点 垂直方向（从上到下，逐列）遍历的值。

如果两个结点在同一行和列，那么顺序则为 **从左到右**。

示例:

示例 1:

输入: [3,9,20,null,null,15,7]

```
    3
   / \
  /   \
 9   20
    / \
   /   \
  15   7
```

输出:

```
[  
  [9],  
  [3,15],  
  [20],  
  [7]  
]
```

```
class Solution:  
    def verticalOrder(self, root) -> [[int]]:  
        from collections import deque, defaultdict  
        if not root:  
            return []  
        queue = deque([(0, root)]) # 不仅存入节点，还存入数字，在一条垂直线上的数字相等  
        lookup = defaultdict(list) # defaultdict(list), defaultdict(int)  
        while queue:  
            idx, node = queue.popleft()  
            lookup[idx].append(node.val) # val先list类型，所有用append  
            if node.left:  
                queue.append((idx - 1, node.left)) # 往左加1  
            if node.right:  
                queue.append((idx + 1, node.right)) # 往右加1  
        return [val for key, val in sorted(lookup.items(), key=lambda x: x[0])]
```

就是利用层次遍历和字典进行不一样的层次遍历。

跟普通层次遍历的区别就是，存储结果用字典，返回结果时，字典里的数据应该按照 key 值排序。

3(0)

9(-1)

20(-1)

15(0)

7(1)

所以从小到大排序： 9 , 3 , 15 , 20 , 7

二叉树中的所有路径

257. 二叉树的所有路径

难度 简单 | 281 收藏 分享 切换为英文 | 关注 | 反馈

给定一个二叉树，返回所有从根节点到叶子节点的路径。

说明: 叶子节点是指没有子节点的节点。

示例:

输入:

```
1
/
2   \
  3
  \
    5
```

输出: ["1->2->5", "1->3"]

解释: 所有根节点到叶子节点的路径为: 1->2->5, 1->3

不能到 None，因为会有两个路径

```
class Solution:
    def binaryTreePaths(self, root):
        if root is None:
            return []
        result = list()
        self.dfs(root, [], result)
        return result

    def dfs(self, node, path, result):
        if node == None:
            return
        if node.left is None and node.right is None: # 不能用node == None, 那样的话路径会出现两遍
            result.append('->'.join(path + [str(node.val)]))
            return
        self.dfs(node.left, path + [str(node.val)], result)
        self.dfs(node.right, path + [str(node.val)], result)
```

判断是否是二叉排序树

98. 验证二叉搜索树

难度 中等 704 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含小于当前节点的数。
- 节点的右子树只包含大于当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

```
输入:  
    2  
   / \  
  1   3  
输出: true
```

示例 2:

```
输入:  
    5  
   / \  
  1   4  
     / \  
    3   6  
输出: false  
解释: 输入为: [5,1,4,null,null,3,6]。  
根节点的值为 5 ，但是其右子节点值为 4 。
```

传入的参数不仅仅是一个根节点，而应该包含最大和最小节点

```
class Solution:  
    def isValidBST(self, root: TreeNode) -> bool:  
        return self.utils(root, None, None) # root不在任何左树下，也不在任何右树下  
  
    def utils(self, root, leftMax, rightMin):  
        if root == None: # 出口，空树是一棵二叉排序树  
            return True  
        if leftMax and root.val >= leftMax.val: # root在leftMax树下，不能超过左树的最大节点  
            return False  
        if rightMin and root.val <= rightMin.val: # 不能小于右树的最小节点  
            return False  
        return self.utils(root.left, root, rightMin) and self.utils(root.right, leftMax, root)
```

二叉排序树的搜索

700. 二叉搜索树中的搜索

难度 简单 80 收藏 分享 切换为英文 关注 反馈

给定二叉搜索树（BST）的根节点和一个值。你需要在BST中找到节点值等于给定值的节点。返回以该节点为根的子树。如果节点不存在，则返回 NULL。

例如，

给定二叉搜索树：



和值： 2

你应该返回如下子树：



在上述示例中，如果要找的值是 5，但因为没有节点值为 5，我们应该返回 NULL。

递归解法：

```
class Solution:
    def searchBST(self, root: TreeNode, val: int) -> TreeNode:
        if not root or root.val == val:
            return root
        if root.val > val:
            return self.searchBST(root.left, val)
        else:
            return self.searchBST(root.right, val)
```

非递归解法：

```
class Solution:
    def searchBST(self, root: TreeNode, val: int) -> TreeNode:
        while root and root.val != val:
            root = root.left if root.val > val else root.right
        return root
```

二叉排序树的插入

701. 二叉搜索树中的插入操作

难度 中等 77 收藏 分享 切换为英文 关注 反馈

给定二叉搜索树（BST）的根节点和要插入树中的值，将值插入二叉搜索树。返回插入后二叉搜索树的根节点。保证原始二叉搜索树中不存在新值。

注意，可能存在多种有效的插入方式，只要树在插入后仍保持为二叉搜索树即可。你可以返回任意有效的结果。

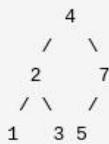
例如，

给定二叉搜索树：

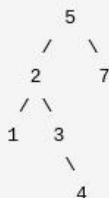


和 插入的值： 5

你可以返回这个二叉搜索树：



或者这个树也是有效的：



二叉树的插入其实和查找没什么区别，因为给定一个节点，进行插入时，你细品一下，一定插入的位置会是叶子节点的。二叉树的插入其实就是构建一棵二叉树的过程，返回的应该是插入好的二叉树的节点。

```
class Solution:
    def insert(self, root, val): # 往tree中插入值为val的节点，并返回插入好的树
        if root == None:
            return TreeNode(val) # 一定会插入叶子节点
        if val < root.val:
            root.left = self.insert(root.left, val)
        else:
            root.right = self.insert(root.right, val)
        return root
```

二叉排序树的删除

450. 删除二叉搜索树中的节点

难度 中等 257 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树的根节点 `root` 和一个值 `key`，删除二叉搜索树中的 `key` 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

- 首先找到需要删除的节点；
- 如果找到了，删除它。

说明：要求算法时间复杂度为 $O(h)$ ， h 为树的高度。

示例：

```
root = [5, 3, 6, 2, 4, null, 7]
key = 3

      5
     / \
    3   6
   / \   \
  2   4   7
```

给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。

一个正确的答案是 `[5, 4, 6, 2, null, null, 7]`，如下图所示。

```
      5
     / \
    4   6
   /     \
  2       7
```

另一个正确答案是 `[5, 2, 6, null, 4, null, 7]`。

```
      5
     / \
    2   6
   / \   \
  4   7
```

当左右孩子只存其一时，只需要用子孩子替换要删除节点即可

当左右孩子都存在时，根据中序遍历，那么可以将该节点的左子树的最大值或者右子树的最小值的节点替换该节点，然后删除左子树的最大值或右子树的最小值，而这两者其实都是叶子节点，所以完美解决。

注：1、当然你需要用到二分查找的方式去查找最小值和最大值。

2、删除其实也是重新构建二叉排序树的过程

```

class Solution:
    def deleteNode(self, root, val): # 删除以root为根节点的树中为val的节点，并返回该树
        if root == None:
            return None
        if root.val > val:
            root.left = self.deleteNode(root.left, val)
        if root.val < val:
            root.right = self.deleteNode(root.right, val)
        # 关键在于找到了该值该如何操作
        if root.val == val:
            if root.left != None and root.right != None:
                maxVal = self.maxVal(root.left)
                root.val = maxVal # 其实并没有删掉该节点，只是替换了值而已
                root.left = self.deleteNode(root.left, maxVal) # 这里用的是左子树的最大值
            elif root.left == None and root.right == None:
                root = None
            elif root.left == None:
                root = root.right
            elif root.right == None:
                root = root.left
        return root

    def maxVal(self, root): # 找root树的最大节点值
        if root.right == None: # 当最右边的有孩子为空时，表明已经找到了最大节点
            return root.val
        val = self.maxVal(root.right)
        return val

```

二叉搜索树的中序后继

从中序遍历的特性去寻找：左-根-右。中序遍历一个结点时，下一个结点有三种情况：

1. 如果当前结点有右结点，则下一个遍历的是右子树的最左结点；
2. 如果当前结点无右结点，若它是父节点的左儿子，则下一遍历的是父节点；
3. 如果当前结点无右结点，且它是父节点的右儿子，则所在子树遍历完了。向上寻找一个作为左儿子的祖先结点，那么下一遍历的就是该祖先结点的父节点；（一直找到根节点为止）
4. 如果上面三种情况都没找到，则该节点是树的最后一个结点，无后继结点；

```

def inorderSuccessor2(self, root, p):
    stack, flag = [], False
    while stack or root:
        if root:
            stack.append(root)
            root = root.left
        else:
            pos = stack.pop()
            if flag:
                return pos
            if pos.val == p.val:
                flag = True
            root = pos.right
    return None

```

二叉搜索树迭代器

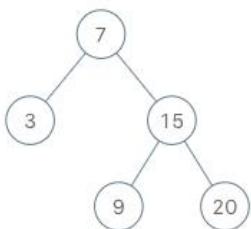
173. 二叉搜索树迭代器

难度 中等 201 收藏 分享 切换为英文 关注 反馈

实现一个二叉搜索树迭代器。你将使用二叉搜索树的根节点初始化迭代器。

调用 `next()` 将返回二叉搜索树中的下一个最小的数。

示例：



```
BSTIterator iterator = new BSTIterator(root);
iterator.next();    // 返回 3
iterator.next();    // 返回 7
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 9
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 15
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 20
iterator.hasNext(); // 返回 false
```

提示：

- `next()` 和 `hasNext()` 操作的时间复杂度是 $O(1)$ ，并使用 $O(h)$ 内存，其中 h 是树的高度。
- 你可以假设 `next()` 调用总是有效的，也就是说，当调用 `next()` 时，BST 中至少存在一个下一个最小的数。

```
class BSTIterator:
    def __init__(self, root):
        self.stack = []
        self.pos = root

    def hasNext(self):
        return self.pos is not None or len(self.stack) > 0 # 只有栈不为空或指针不指向空就有下一个节点

    def next(self): # 非递归中序遍历的分解应用，调用一次该函数表示将求出一个中序后继
        # 注意这里因为经常反复调用了next函数，所以略去了最外层循环，但是往左边递归找空节点是不能少的，所以这里由if变成了while
        while self.pos:
            self.stack.append(self.pos)
            self.pos = self.pos.left
        self.pos = self.stack.pop()
        nxt = self.pos
        self.pos = self.pos.right
        return nxt.val
```

将有序数组转化为二叉搜索树

108. 将有序数组转换为二叉搜索树

难度 简单 499 收藏 分享 切换为英文 关注 反馈

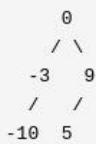
将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。

示例：

给定有序数组：[-10, -3, 0, 5, 9]，

一个可能的答案是：[0, -3, 9, -10, null, 5]，它可以表示下面这个高度平衡二叉搜索树：



```
class Solution:
    def sortedArrayToBST(self, nums: [int]) -> TreeNode:
        if len(nums) == 0: # 出口是当数组为空时，转化为空节点
            return None
        mid = len(nums) // 2
        root = TreeNode(nums[mid])
        root.left = self.sortedArrayToBST(nums[:mid])
        root.right = self.sortedArrayToBST(nums[mid + 1:])
        return root
```

修剪二叉搜索树

669. 修剪二叉搜索树

难度 简单 231 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树，同时给定最小边界 L 和最大边界 R 。通过修剪二叉搜索树，使得所有节点的值在 $[L, R]$ 中 ($R \geq L$)。你可能需要改变树的根节点，所以结果应当返回修剪好的二叉搜索树的新的根节点。

示例 1:

输入:

```
1
/\ 
0  2
```

```
L = 1
R = 2
```

输出:

```
1
 \
 2
```

示例 2:

输入:

```
3
/\ 
0  4
 \
 2
 /
1
```

```
L = 1
R = 3
```

输出:

```
3
/
2
/
1
```

```
class Solution:
    def trimBST(self, root, minimum, maximum): # 返回符合条件的那棵树
        if not root: # 出口，节点为空时，返回该节点
            return None
        if root.val < minimum: # 根节点值小于最小值，那么根节点以及左树都将小于，抛弃根节点及其左树
            return self.trimBST(root.right, minimum, maximum)
        if root.val > maximum:
            return self.trimBST(root.left, minimum, maximum)
        root.left = self.trimBST(root.left, minimum, maximum) # 根节点满足条件，继续遍历，左右都不能抛弃
        root.right = self.trimBST(root.right, minimum, maximum)
        return root
```

相同的树

100. 相同的树

难度 简单 386 收藏 分享 切换为英文 关注 反馈

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

示例 1：

输入：
 1 1
 / \ / \
 2 3 2 3

[1, 2, 3], [1, 2, 3]

输出：true

示例 2：

输入：
 1 1
 / \
 2 2

[1, 2], [1, null, 2]

输出：false

示例 3：

输入：
 1 1
 / \ / \
 2 1 1 2

[1, 2, 1], [1, 1, 2]

输出：false

两棵树的左右子树相等，那么他们就会相等

```
class Solution:  
    def isSameTree(self, a, b): # 比较两个树是否相等  
        if a == None and b == None:  
            return True  
        if a == None or b == None:  
            return False  
        if a.val != b.val:  
            return False  
        a1 = self.isSameTree(a.left, b.left)  
        b1 = self.isSameTree(a.right, b.right)  
        return a1 and b1 # 节点相等，左树相等，右树相等，那么两棵树相等
```

树的子结构

剑指 Offer 26. 树的子结构

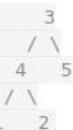
难度 中等 64 收藏 分享 切换为英文 关注 反馈

输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)

B是A的子结构，即 A中有出现和B相同的结构和节点值。

例如：

给定的树 A：



给定的树 B：



返回 true，因为 B 与 A 的一个子树拥有相同的结构和节点值。

示例 1：

```
输入：A = [1,2,3], B = [3,1]
输出：false
```

示例 2：

```
输入：A = [3,4,5,1,2], B = [4,1]
输出：true
```

限制：

0 <= 节点个数 <= 10000

层次遍历加 + 相同树比较

```
class Solution:
    def isSubStructure(self, pRoot1, pRoot2):      # pRoot2树是否是pRoot1树的子结构
        if not pRoot2 or not pRoot1:    # 空树不是任何树的子结构
            return False
        queue = list()
        queue.append(pRoot1)
        flag = False
        while queue:
            pos = queue.pop(0)
            if pos.val == pRoot2.val: # 相等，开始检查是否是子结构
                flag = self.is_subTree(pos, pRoot2)
                if flag == True:
                    return True
            if pos.left:
                queue.append(pos.left)
            if pos.right:
                queue.append(pos.right)
        return flag

    def is_subTree(self, pRoot1, pRoot2):
        if not pRoot2:  # 从树到了叶子节点，表明含子结构
            return True
        if not pRoot1:  # 从树没到叶子结点，主树到了，返回False
            return False
        if pRoot1.val != pRoot2.val:
            return False
        else: # 当根节点相等时，比较子节点是否相等。
            return self.is_subTree(pRoot1.left, pRoot2.left) and self.is_subTree(pRoot1.right, pRoot2.right)
```

检查子树

面试题 04.10. 检查子树

难度 中等 13 收藏 分享 切换为英文 关注 反馈

检查子树。你有两棵非常大的二叉树：T1，有几万个节点；T2，有几万个节点。设计一个算法，判断 T2 是否为 T1 的子树。

如果 T1 有这么一个节点 n，其子树与 T2 一模一样，则 T2 为 T1 的子树，也就是说，从节点 n 处把树砍断，得到的树与 T2 完全相同。

示例1：

```
输入 : t1 = [1, 2, 3], t2 = [2]
输出 : true
```

示例2：

```
输入 : t1 = [1, null, 2, 4], t2 = [3, 2]
输出 : false
```

提示：

1. 树的节点数目范围为[0, 20000]。

解法一：遍历两棵树，看 T2 的前序遍历序列是否是 T 的前序遍历序列，记得用 find，别用 index。

```
class Solution:
    def pre_order(self, root, rt):
        if not root:
            return
        rt.append(root.val)
        self.pre_order(root.left, rt)
        self.pre_order(root.right, rt)

    def checkSubTree(self, T1, T2):
        rt = []
        self.pre_order(T1, rt)
        string = "".join(str(x) for x in rt)
        rt2 = []
        self.pre_order(T2, rt2)
        string2 = "".join(str(x) for x in rt2)
        return string.find(string2) != -1 # 看T1的前序遍历序列是否包含T2的前序遍历序列
```

解法二：跟树的子结构一个思路

```

class Solution:
    def checkSubTree(self, pRoot1, pRoot2):      # pRoot2树是否是pRoot1树的子结构
        if not pRoot2 or not pRoot1:  # 空树不是任何树的子结构
            return False
        queue = []
        queue.append(pRoot1)
        flag = False
        while queue:
            pos = queue.pop(0)
            if pos.val == pRoot2.val: # 相等，开始检查是否是子结构
                flag = self.is_subTree(pos, pRoot2)
                if flag == True:
                    return True
            if pos.left:
                queue.append(pos.left)
            if pos.right:
                queue.append(pos.right)
        return flag

    def is_subTree(self, pRoot1, pRoot2):
        if not pRoot2 and not pRoot1: # 同时到达叶子节点才为真
            return True
        if pRoot1.val != pRoot2.val:
            return False
        else: # 当根节点相等时，比较子节点是否相等。
            return self.is_subTree(pRoot1.left, pRoot2.left) and self.is_subTree(pRoot1.right, pRoot2.right)

```

存的是区间和该树的最大值

构造二叉树

构造平衡二叉树

```

def build(self, ans, l, r): # 利用二分法建立平衡二叉树，其中 ans 是升序的

    if l == r:
        return TreeNode(self.ans[l]) # 建立叶子结点

    if l > r:
        return
    mid = (l + r) // 2

    node = TreeNode(self.ans[mid]) # 建立根节点和非叶子节点

    node.left = self.build(l, mid - 1)
    node.right = self.build(mid + 1, r)

    return node # 返回一根子树

```

构造普通二叉树

```

def buildTree(self, start, A): # 从 start 处开始构造普通二叉树，一般从 0 开始，A
    无序

    if start >= len(A):
        return
    node = TreeNode(A[start])

    node.left = self.buildTree(2 * start + 1, A) # 下一个节点为 2 * start + 1
    node.right = self.buildTree(2 * start + 2, A)
    return node

```

根据前序和中序遍历重建二叉树

题目描述

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

根据前序遍历找到根节点，根据根节点将中序遍历序列分为两半。

函数的意义是：根据前序和中序遍历序列重建二叉树。当序列为空时，空姐点就是重构的一颗二叉树。

```

class Solution:
    # 返回构造的TreeNode根节点
    def reConstructBinaryTree(self, pre, tin):
        if len(tin) <= 0:
            return None
        root = TreeNode(pre[0])
        pos = tin.index(pre[0])
        root.left = self.reConstructBinaryTree(pre[1:pos+1], tin[:pos])
        root.right = self.reConstructBinaryTree(pre[pos+1:], tin[pos+1:])
        return root

```

前、中、后遍历递归与非递归实现：

前序递归遍历：

```

def pre_order1(self, root, result):
    if root == None:
        return
    result.append(root.val)
    self.pre_order1(root.left, result)
    self.pre_order1(root.right, result)
    return result

```

前序遍历非递归实现：

```
def pre_order2(self, root):
    result = []
    stack = []
    while root or stack:
        if root:
            result.append(root.val) # 前序遍历点
            stack.append(root)
            root= root.left
        else:
            root= stack.pop()
            root= root.right
    return result
```

中序遍历递归实现：

```
def mid_order1(self, root, result):
    if root == None:
        return
    self.mid_order1(root.left, result)
    result.append(root.val)
    self.mid_order1(root.right, result)
```

中序遍历非递归实现：

```
def mid_order2(self, root):
    result = []
    stack = []
    while root or stack:
        if root is not None:
            stack.append(root)
            root = root.left
        else:
            root = stack.pop()
            result.append(root.val)
            root = root.right
    return result
```

后序遍历递归实现：

```

def post_order1(self, root, result):
    if root == None:
        return
    self.post_order1(root.left, result)
    self.post_order1(root.right, result)
    result.append(root.val)

```

后序遍历非递归实现：

```

def post_order2(self, root):
    stack, res = [], []
    stack.append(root)
    while stack:
        root = stack.pop() # 唯一和层次遍历不同的地方
        res.append(root.val)
        if root.left:
            stack.append(root.left)
        if root.right:
            stack.append(root.right)
    return res[::-1] # 层次遍历改成pop(),然后结果逆序

```

BFS 遍历实现

```

def BFS(self, root):
    result = []
    queue = collections.deque()
    queue.append(root)
    while queue:
        root = queue.popleft()
        result.append(root.val)
        if root.left is not None:
            queue.append(root.left)
        if root.right is not None:
            queue.append(root.right)
    return result

```

```

class Solution:
    def levelOrder(self, root):
        if root is None:
            return []
        queue = deque([root])
        result = []
        while queue:
            level = [] # 记录一层的节点
            for _ in range(len(queue)): # 该层的个数由queue中的个数决定
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            result.append(level)
        return result

```

包含 null 的 BFS 实现：

```

def serialize(self, root): # 返回root树的遍历序列
    queue, data = [], []
    queue.append(root)
    while queue:
        pos = queue.pop(0)
        if pos != None:
            data.append(pos.val)
            queue.append(pos.left)
            queue.append(pos.right)
        else:
            data.append("None")
    return data

```

求最大深度

```

def max_depth(self, root):
    if root == None:
        return 0
    ldepth = self.max_depth(root.left)
    rdepth = self.max_depth(root.right)

    return max(ldepth, rdepth) + 1

```

求所有节点数

```
def nums(self, root):
    if root == None:
        return 0

    lnums = self.nums(root.left)
    rnums = self.nums(root.right)

    return lnums + rnums + 1
```

链表

链表求和

面试题 02.05. 链表求和

难度 中等 26 收藏 分享 切换为英文 关注 反馈

给定两个用链表表示的整数，每个节点包含一个数位。

这些数位是反向存放的，也就是个位排在链表首部。

编写函数对这两个整数求和，并用链表形式返回结果。

示例：

```
输入：(7 -> 1 -> 6) + (5 -> 9 -> 2)，即617 + 295
输出：2 -> 1 -> 9，即912
```

进阶：假设这些数位是正向存放的，请再做一遍。

示例：

```
输入：(6 -> 1 -> 7) + (2 -> 9 -> 5)，即617 + 295
输出：9 -> 1 -> 2，即912
```

通过次数 10,247 提交次数 22,487

在真实的面试中遇到过这道题？ 是 否

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        newHead, carry = ListNode(0), 0
        start = newHead
        while l1 or l2 or carry:
            a = l1.val if l1 else 0
            b = l2.val if l2 else 0
            cur = (a + b + carry) % 10
            carry = (a + b + carry) // 10
            start.next = ListNode(cur)
            start = start.next
            if l1: l1 = l1.next
            if l2: l2 = l2.next
        return newHead.next
```

两数相加 2

445. 两数相加 II

难度 中等 222 收藏 分享 切换为英文 关注 反馈

给你两个 非空 链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储一位数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

进阶：

如果输入链表不能修改该如何处理？换句话说，你不能对列表中的节点进行翻转。

示例：

```
输入：(7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 8 -> 0 -> 7
```

因为两者的位数可能不同，所以变得有点难搞，所以必须逆序，那么就可以借助栈来实现逆序。假如长度不一致之后的话，少的那部分因子补 0 即可。所以循环跳出的条件应该是 stack1 or stack2 or carry != 0，而且应该采用从前往后重新构造链表的方式，而不是从后往前重新构造链表。这种进位的思想值得学习。永远求的都是进位本身，而不是进位的下一位。

```

class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        tail, curNode = None, None # 尾部已相加的链表、当前正在相加的节点
        carry = 0
        if l1.val == 0:
            return l2
        if l2.val == 0:
            return l1
        stack1, stack2 = list(), list()
        while l1:
            stack1.append(l1)
            l1 = l1.next
        while l2:
            stack2.append(l2)
            l2 = l2.next
        while stack1 or stack2 or carry != 0:
            a = stack1.pop().val if stack1 else 0 # 第一个加数
            b = stack2.pop().val if stack2 else 0 # 第二个加数
            curSum = a + b + carry # 和
            curVal = curSum % 10 # 当前位
            carry = curSum // 10 # 进位
            curNode = ListNode(curVal)
            curNode.next = tail
            tail = curNode
        return curNode

```

分割链表

86. 分隔链表

难度 中等 233 收藏 分享 切换为英文 关注 反馈

给定一个链表和一个特定值 x ，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。

你应当保留两个分区中每个节点的初始相对位置。

示例:

```

输入: head = 1->4->3->2->5->2, x = 3
输出: 1->2->2->4->3->5

```

思路：建立两个新链表，都含有头尾指针，分割后连接两个链表即可，注意移动的是两个新链表的尾指针，等到遍历完成，前面链表的尾指针要指向后面链表的头节点，后面链表的头指针要指向空

```
class Solution:
    def partition(self, head: ListNode, x: int) -> ListNode:
        preHead, postHead, start = ListNode(0), ListNode(0), head
        preTail, postTail = preHead, postHead
        while start:
            if start.val < x:
                preTail.next = start
                preTail = preTail.next
            else:
                postTail.next = start
                postTail = postTail.next
            start = start.next # 链表循环进行条件不能忘
        postTail.next = None # 链表末尾一定要指向空
        preTail.next = postHead.next
        return preHead.next
```

删除链表中的重复元素 I

83. 删除排序链表中的重复元素

难度 简单 357 收藏 分享 切换为英文 关注 反馈

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1：

```
输入: 1->1->2
输出: 1->2
```

示例 2：

```
输入: 1->1->2->3->3
输出: 1->2->3
```

注意：用到元素的下下个元素时，循环跳出条件是：start and start.next

```
class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        start = head
        while start and start.next:
            if start.val == start.next.val:
                start.next = start.next.next
            else:
                start = start.next
```

删除链表中的重复元素 II

82. 删除排序链表中的重复元素 II

难度 中等 327 收藏 分享 切换为英文 关注 反馈

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 没有重复出现 的数字。

示例 1：

```
输入: 1->2->3->3->4->4->5  
输出: 1->2->5
```

示例 2：

```
输入: 1->1->1->2->3  
输出: 2->3
```

```
class Solution:  
    def deleteDuplicates(self, head: ListNode) -> ListNode:  
        newHead = ListNode(0)  
        newHead.next = head  
        pre, start = newHead, head  
        while start and start.next:  
            flag = False  
            while start.next and start.val == start.next.val:  
                start.next = start.next.next  
                flag = True  
            if flag: # 上面还保留重复元素中的第一个, 这一步可以干掉它  
                pre.next = start.next  
                start = pre.next  
            else:  
                pre = pre.next  
                start = pre.next  
        return newHead.next
```

反转链表

206. 反转链表

难度 简单 1095 收藏 分享 切换为英文 关注 反馈

反转一个单链表。

示例：

```
输入: 1->2->3->4->5->NULL  
输出: 5->4->3->2->1->NULL
```

进阶：

你可以迭代或递归地反转链表。你能否用两种方法解决这道题？

迭代法：

```

class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        newHead = ListNode(0)
        newHead.next = head
        start = head
        while start and start.next:# start只能到最后一个元素，不能到None
            temp = start.next
            start.next = temp.next
            temp.next = newHead.next
            newHead.next = temp
        return newHead.next

```

递归法：

```

class Solution:
    def reverseList(self, head: ListNode) -> ListNode: # 反转链表，并返回反转表尾节点
        if head == None or head.next == None: return head
        tail = self.reverseList(head.next)
        head.next.next = head
        head.next = None # 备记，不然会形成循环链表
        return tail

```

精髓在于当到达尾部节点时，返回该节点，之后一直返回该节点，然后不断翻转。

K 组翻转链表

25. K 个一组翻转链表

难度 困难 614 收藏 分享 切换为英文 关注 反馈

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

示例：

给你这个链表：1->2->3->4->5

当 $k = 2$ 时，应当返回：2->1->4->3->5

当 $k = 3$ 时，应当返回：3->2->1->4->5

说明：

- 你的算法只能使用常数的额外空间。
- 你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

```

class Solution:
    def reverse(self, start, end):
        newHead = ListNode(0) # 做链表题时首先定义一个头节点，指向链表
        newHead.next = start
        while start.next != end: # start一直往后走，直到找到end
            temp = start.next
            start.next = temp.next
            temp.next = newHead.next
            newHead.next = temp
        start.next = end.next # 找到end节点后，将其置于头节点
        end.next = newHead.next
        newHead.next = end
        return end, start # 返回翻转后的链表的头结点和下一个范围的起始点的前驱

    def reverseKGroup(self, head, k):
        if k == 1:
            return head
        newHead = ListNode(0)
        newHead.next = head
        start = newHead # 好计算k个链表节点以及衔接翻转好的链表
        while start.next:
            end = start # end是基于start开始往后算的
            for i in range(k):
                end = end.next
                if end == None: # 当不满足k个数时，结束翻转
                    return newHead.next
            doneHead, nextStart = self.reverse(start.next, end)
            start.next = doneHead
            start = nextStart # 下一个翻转链表的前驱节点
        return newHead.next

```

Reverse 中的 while 循环条件不能是 start!= end , 因为这表示 start == end , 那么假如 end 是最后一个节点 ,那么 temp = start.next = None , 那么 temp.next 会报错 ,所以只能 start.next == end 就跳出循环。

这道题的精髓在于 :

传入的节点是 k 组链表的头和尾 , 返回的是翻转好的头和尾。

合并 k 个排序链表

23. 合并K个排序链表

难度 困难 击 790 收藏 分享 切换为英文 关注 反馈

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

示例:

输入:
[
 1->4->5,
 1->3->4,
 2->6
]
输出: 1->1->2->3->4->4->5->6

归并排序其实也是合并 k 个数组的过程 ,只不过这 k 个数组是由一个数组拆开的 ,所以结合归并排序 ,转化为合并两个排序链表的算法。

```

class Solution:
    def mergeKLists(self, lists: [ListNode]) -> ListNode: # 合并k个排序链表
        if len(lists) == 0:
            return None
        left, right = 0, len(lists) - 1
        return self.mergeK(lists, left, right)

    def mergeK(self, lists, left, right): # 合并k个链表其实就是对大小为k的数组进行归并排序
        if left == right:
            return lists[left]
        mid = (left + right) // 2
        leftNode = self.mergeK(lists, left, mid)
        rightNode = self.mergeK(lists, mid + 1, right)
        return self.merge(leftNode, rightNode)

    def merge(self, leftNode, rightNode): # 递归合并两个有序链表
        if not leftNode: # 当有一个链表为空时就是出口
            return rightNode
        if not rightNode:
            return leftNode
        if leftNode.val < rightNode.val:
            leftNode.next = self.merge(leftNode.next, rightNode)
            return leftNode # 谁小就返回谁的节点，当然该节点后面应该跟上由它开头的新的递归
        else:
            rightNode.next = self.merge(leftNode, rightNode.next)
            return rightNode

```

链表的中间节点

876. 链表的中间结点

难度 简单 229 收藏 分享 切换为英文 关注 反馈

给定一个带有头结点 head 的非空单链表，返回链表的中间结点。

如果有两个中间结点，则返回第二个中间结点。

示例 1：

输入：[1, 2, 3, 4, 5]
 输出：此列表中的结点 3 (序列化形式：[3, 4, 5])
 返回的结点值为 3。 (测评系统对该结点序列化表述是 [3, 4, 5])。
 注意，我们返回了一个 ListNode 类型的对象 ans，这样：
`ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, 以及 ans.next.next.next = NULL.`

示例 2：

输入：[1, 2, 3, 4, 5, 6]
 输出：此列表中的结点 4 (序列化形式：[4, 5, 6])
 由于该列表有两个中间结点，值分别为 3 和 4，我们返回第二个结点。

定义快慢指针，快指针的速度是慢指针的两倍。

注意防止 fast.next 空指针异常，首先要判断 fast 不为 nil

```
class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        fast, low = head, head
        while fast and fast.next:
            fast = fast.next.next
            low = low.next
        return low
```

删除链表中的倒数第 N 个节点

19. 删除链表的倒数第N个节点

难度 中等 862 收藏 分享 切换为英文 关注 反馈

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

给定一个链表：1->2->3->4->5，和 $n = 2$ 。

当删除了倒数第二个节点后，链表变为 1->2->3->5。

说明：

给定的 n 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

利用快慢指针，快指针先走 N 个节点，慢指针后走，那么快指针到末尾后，慢指针所指位置即是倒数第 N 个节点。

这里值得注意的一点就是，如果快指针走了 N 个节点为空了，就表明第一个节点就是要删除的节点。

```
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        fast, low = head, head
        for i in range(n):
            fast = fast.next
        if fast == None:
            head = head.next # 第一个就是要删除的节点
            return head
        # fast到最后一个节点时，low到达倒数第k个，而不是当fast到达空节点时
        while fast and fast.next:
            fast = fast.next
            low = low.next
        low.next = low.next.next
        return head
```

奇偶链表

328. 奇偶链表

难度 中等 206 收藏 分享 切换为英文 关注 反馈

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 $O(1)$ ，时间复杂度应为 $O(\text{nodes})$ ， nodes 为节点总数。

示例 1：

```
输入: 1->2->3->4->5->NULL
输出: 1->3->5->2->4->NULL
```

示例 2：

```
输入: 2->1->3->5->6->4->7->NULL
输出: 2->3->6->7->1->5->4->NULL
```

说明：

- 应当保持奇数节点和偶数节点的相对顺序。
- 链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

如果这道题不需要原地的话，那毫无疑问就是定义两个链表，四个指针，但因为是原地，头结点可以替代奇链表头指针，故只需要定义三个：指向下一个奇数索引的指针 odd，指向偶数索引的指针 even 和指向偶数链表头部的指针 even_head。

```
class Solution:
    def oddEvenList(self, head: ListNode) -> ListNode:
        if head == None:
            return None
        odd, even, evenHead = head, head.next, head.next
        while even and even.next:# even为空或者even的下一个元素为空时，表明没有了奇数节点
            odd.next = even.next
            odd = odd.next
            even.next = odd.next
            even = even.next
        odd.next = evenHead
        return head
```

环形链表

定义快慢两个指针，快指针的速度是慢指针的两倍，如果快指针追上了慢指针，表明，有环。

```

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        low, fast = head, head
        while fast and fast.next:
            fast = fast.next.next
            low = low.next
            if fast == low:
                return True
        return False

```

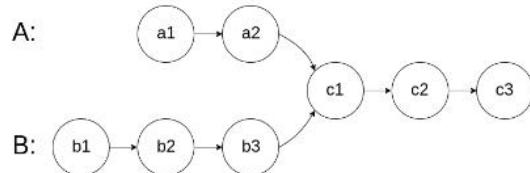
相交链表

160. 相交链表

难度 简单 721 收藏 分享 切换为英文 关注 反馈

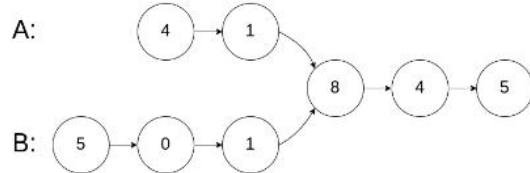
编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

示例 1：



输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8

输入解释：相交节点的值为 8（注意，如果两个链表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

该题要求从两链表的表头算起，所以增加了难度，需计算出二者的的长度差，然后长链表先走完该长度差，然后计算与短链表是否有相交的地方。但是这个长度差要计算两链表的长度就会消耗额外的时间复杂度，所以该题目有一种巧妙的解法，如下：

解题思路：

- 我们通常做这种题的思路是设定两个指针分别指向两个链表头部，一起向前走直到其中一个到达末端，另一个与末端距离则是两链表的 长度差。再通过长链表指针先走的方式消除长度差，最终两链表即可同时走到相交点。

- 换个方式消除长度差：拼接两链表。**

设长-短链表为 C，短-长链表为 D（分别代表长链表在前和短链表在前的拼接链表），则当 C 走到长短链表交接处时，D 走在长链表中，且与长链表头距离为 长度差；

以下图片帮助理解：当 `ha == hb` 时跳出，返回即可



```
class Solution:  
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:  
        ha, hb = headA, headB  
        while ha != hb:  
            ha = ha.next if ha else headB # 这里不应该为hb  
            hb = hb.next if hb else headA  
        return ha
```

图

图的基本概念

图分两种：

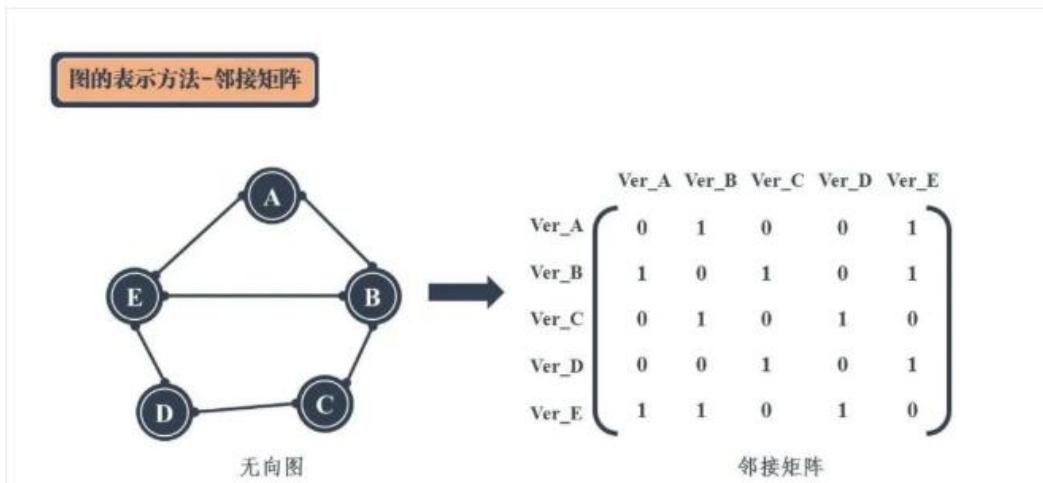
有向图（节点有入度和出度）、无环图（可以间接认为只有出度）

图有两种存储方式：（我的代码实现的是简单的邻接表）

A、邻接矩阵：

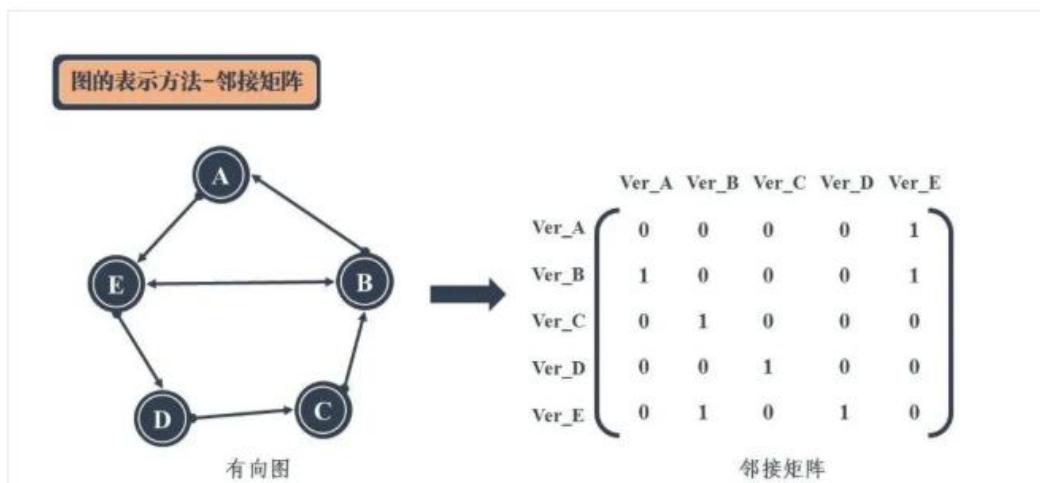
无向图：

目前常用的图存储方式为邻接矩阵，通过所有顶点的二维矩阵来存储两个顶点之间是否相连，或者存储两顶点间的边权重。



无向图的邻接矩阵是一个对称矩阵，是因为边不具有方向性，若能从此顶点能够到达彼顶点，那么彼顶点自然也能够达到此顶点。此外，由于顶点本身与本身相连没有意义，所以在邻接矩阵中对角线上皆为0。

有向图：



有向图由于边具有方向性，因此彼此顶点之间并不能相互达到，所以其邻接矩阵的对称性不再。

用邻接矩阵可以直接从二维关系中获得任意两个顶点的关系，可直接判断是否相连。但是在对矩阵进行存储时，却需要完整的一个二维数组。若图中顶点数过多，会导致二维数组的大小剧增，从而占用大量的内存空间。

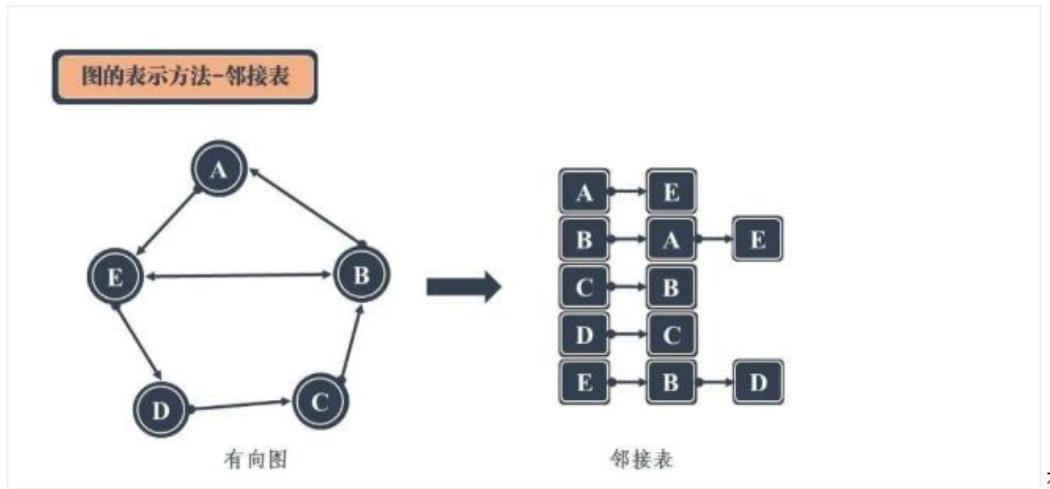
而根据实际情况可以分析得，图中的顶点并不是任意两个顶点间都会相连，不是都需要对其边上权重进行存储。那么存储的邻接矩阵实际上会存在大量的0。虽然可以通过稀疏表示等方式对稀疏性高的矩阵进行关键信息的存储，但是却增加了图存储的复杂性。

因此，为了解决上述问题，一种可以只存储相连顶点关系的邻接表应运而生。

B、邻接表

邻接表

在邻接表中，图的每一个顶点都是一个链表的头节点，其后连接着该顶点能够直接达到的相邻顶点。相较于无向图，有向图的情况更为复杂，因此这里采用有向图进行实例分析。



在

邻接表中，每一个顶点都对应着一条链表，链表中存储的是顶点能够达到的相邻顶点。存储的顺序可以按照顶点的编号顺序进行。比如上图中对于顶点B来说，其通过有向边可以到达顶点A和顶点E，那么其对应的邻接表中的顺序即B->A->E，其它顶点亦如此。

通过邻接表可以获得从某个顶点出发能够到达的顶点，从而省去了对不相连顶点的存储空间。然而，这还不够。对于有向图而言，图中有效信息除了从顶点“指出去”的信息，还包括从别的顶点“指进来”的信息。这里的“指出去”和“指进来”可以用出度和入度来表示。

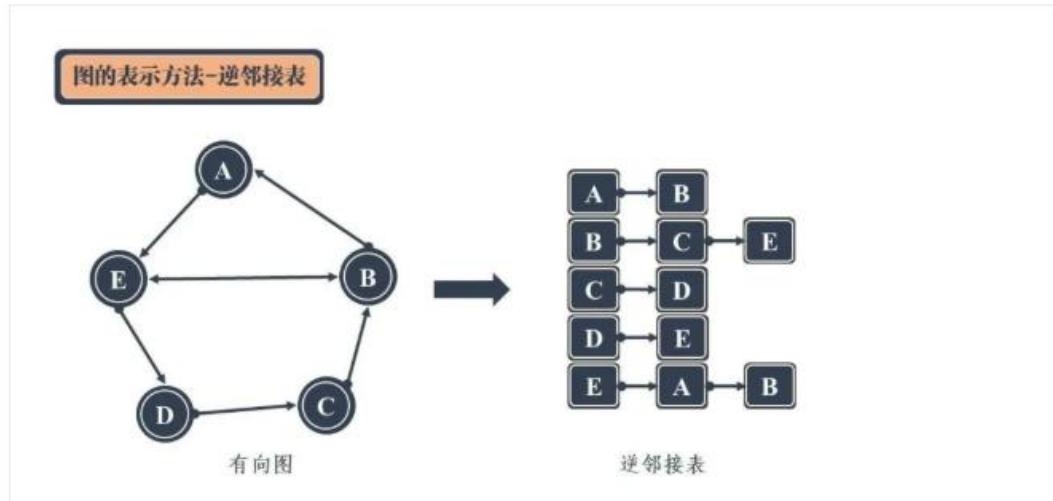
入度：有向图的某个顶点作为终点的次数和。

出度：有向图的某个顶点作为起点的次数和。

由此看出，在对有向图进行表示时，邻接表只能求出图的出度，而无法求出入度。这个问题很好解决，那就是增加一个表用来存储能够到达某个顶点的相邻顶点。这个表称作逆邻接表。

逆邻接表

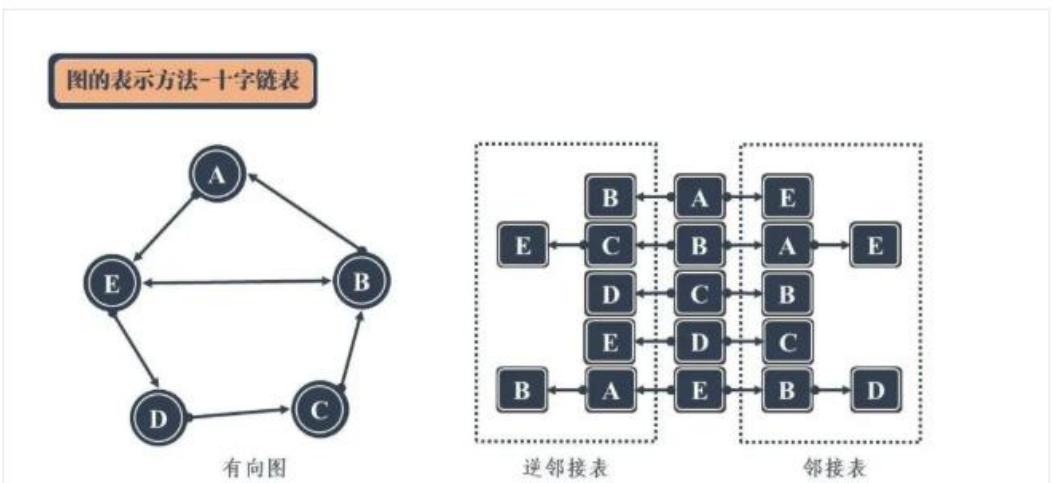
逆邻接表与邻接表结构类似，只不过图的顶点链接着能够到达该顶点的相邻顶点。也就是说，邻接表时顺着图中的箭头寻找相邻顶点，而逆邻接表时逆着图中的箭头寻找相邻顶点。



邻接表和逆邻接表的共同使用下，就能够把一个完整的有向图结构进行表示。可以发现，邻接表和逆邻接表实际上有一部分数据时重合的，因此可以将两个表合二为一，从而得到了所谓的十字链表。

十字链表

十字链表似乎很简单，只需要通过相同的顶点分别链向以该顶点为终点和起点的相邻顶点即可。



但这并不是最优的表示方式。虽然这样的方式共用了中间的顶点存储空间，但是邻接表和逆邻接表的链表节点中重复出现的顶点并没有得到重复利用，反而是进行了再次存储。因此，上图的表示方式还可以进行进一步优化。

十字链表优化后，可通过扩展的顶点结构和边结构来进行正逆邻接表的存储：（下面的弧头可看作是边的箭头那端，弧尾可看作是边的圆点那端）

data: 用于存储该顶点中的数据；

firstin指针：用于连接以当前顶点为弧头的其他顶点构成的链表，即从别的顶点指进来的顶点；

firstout指针：用于连接以当前顶点为弧尾的其他顶点构成的链表，即从该顶点指出去的顶点；

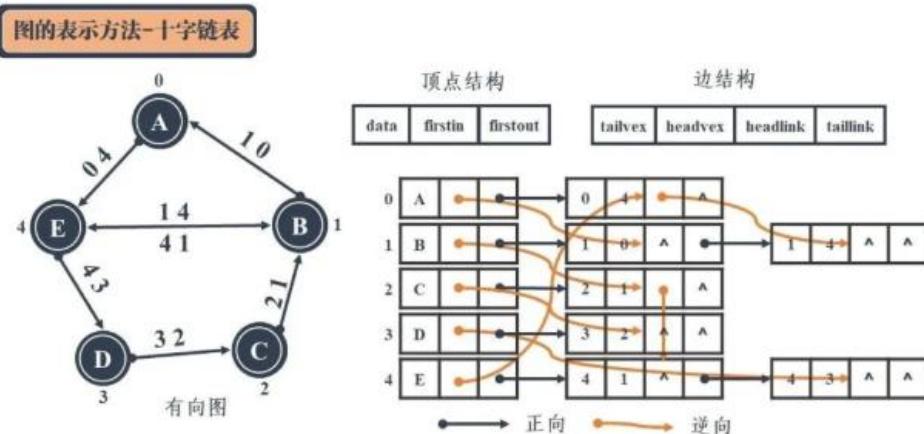
边结构通过存储两个顶点来确定一条边，同时通过分别代表这两个顶点的指针来与相邻顶点进行链接：

tailvex: 用于存储作为弧尾的顶点的编号；

headvex: 用于存储作为弧头的顶点的编号；

headlink 指针：用于链接下一个存储作为弧头的顶点的节点；

taillink 指针：用于链接下一个存储作为弧尾的顶点的节点；



图的结构及其遍历

图的 dfs 遍历：

1、dfs 函数规划：Def dfs(self, point, visit)，遍历以 point 为起点的剩余图结构，返回数组。

2、Base case : if point.val in visit: return

3、状态转移 :

```
Visit.append(pos.val)
For pos in point.outgoing:
    Self.dfs(pos, visit)
```

图的 bfs 遍历 :

```
Def bfs(self, point):
    Import collections
    Queue = collections.deque()
    Queue.append(point)
    Visited, result = [], []
    While queue:
        Pos = queue.popleft()
        Result.append(pos.val)
        Visit.append(pos)
        For outgoing in point.outgoings:
            If outgoing not in visit:
                Queue.append(outgoing)
    Return result
```

```
class point(object):
    def __init__(self, val, outgoing=[]): # outgoing为节点的出度集合，深的是点
        self.val = val
        self.outgoing = outgoing

class Graph:
    def DFS(self, point, visited_points): # 模拟二叉树的深度遍历
        if point in visited_points: # 因为不可能为空
            return
        visited_points.append(point)
        for point in point.outgoing: # 模拟遍历左右子树
            self.DFS(point, visited_points)

    def BFS(self, point):
        res, queue, visited_points = [], [], []
        queue.append(point)
        visited_points.append(point)
        while queue:
            level = []
            length = len(queue)
            for _ in range(length):
                pos = queue.pop(0)
                level.append(pos.val)
                for e in pos.outgoing:
                    if e not in visited_points: # 模拟遍历左右子树
                        queue.append(e)
                        visited_points.append(e)
            res.extend(level)
        return res
```

单词接龙

127. 单词接龙

难度 中等 347 收藏 分享 切换为英文 关注 反馈

给定两个单词 (*beginWord* 和 *endWord*) 和一个字典，找到从 *beginWord* 到 *endWord* 的最短转换序列的长度。转换需遵循如下规则：

1. 每次转换只能改变一个字母。
2. 转换过程中的中间单词必须是字典中的单词。

说明:

- 如果不存在这样的转换序列，返回 0。
- 所有单词具有相同的长度。
- 所有单词只由小写字母组成。
- 字典中不存在重复的单词。
- 你可以假设 *beginWord* 和 *endWord* 是非空的，且二者不相同。

示例 1:

输入:
`beginWord = "hit",
endWord = "cog",
wordList = ["hot", "dot", "dog", "lot", "log", "cog"]`

输出: 5

解释: 一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog",
返回它的长度 5。

示例 2:

输入:
`beginWord = "hit"
endWord = "cog"
wordList = ["hot", "dot", "dog", "lot", "log"]`

输出: 0

解释: *endWord* "cog" 不在字典中，所以无法进行转换。

将其转化为邻接表，然后利用图的 BFS，因为是 BFS，所以当遇到最终节点时，就会使最短路径。但是会超出时间限制，希望以后能找到解决方法

```

class Solution(object):
    def ladderLength(self, beginWord, endWord, wordList):
        neighborList = dict()      # 邻接表是字典类型，key存节点，value存出度点集合
        headList = copy.deepcopy(wordList) # 深拷贝，互不影响
        headList.insert(0, beginWord)
        for head in headList:      # keys
            if head == endWord:
                continue
            tempList = []
            for node in wordList:
                if self.diff(head, node): # 字母相差1位并且之前没用到过，但beginword不算
                    tempList.append(node)
            if len(tempList) != 0: # 为空的话没意义
                neighborList[head] = tempList
        queue = collections.deque()
        queue.append(beginWord)
        minLen = 1
        visit = [beginWord]
        while queue:
            length = len(queue)
            for _ in range(length):
                pos = queue.popleft()
                if pos == endWord: # bfs遍历到目标点，就是最短路径
                    return minLen
                if neighborList.get(pos): # 必须判断是否存在，否则调用器报keyError
                    for node in neighborList.get(pos):
                        if node not in visit:
                            queue.append(node)
                            visit.append(node)
            minLen += 1
        return 0

    def diff(self, word1, word2):
        diff = 0
        for i in range(len(word1)):
            if word1[i] != word2[i]:
                diff += 1
        return True if diff == 1 else False

```

待：847. 访问所有节点的最短路径

847. 访问所有节点的最短路径

难度 困难 75 收藏 分享 切换为英文 关注 反馈

给出 `graph` 为有 N 个节点（编号为 $0, 1, 2, \dots, N-1$ ）的无向连通图。

`graph.length = N`，且只有节点 i 和 j 连通时， $j \neq i$ 在列表 `graph[i]` 中恰好出现一次。

返回能够访问所有节点的最短路径的长度。你可以在任一节点开始和停止，也可以多次重访节点，并且可以重用边。

示例 1：

输入：`[[1,2,3],[0],[0],[0]]`
输出：4
解释：一个可能的路径为 `[1,0,2,0,3]`

示例 2：

输入：`[[1],[0,2,4],[1,3,4],[2],[1,2]]`
输出：4
解释：一个可能的路径为 `[0,1,4,2,3]`

方法一：广度优先搜索【通过】

思路

路径 `state` 表示当前节点和已访问节点的子集。此问题可以简化为 `state` 的最短路径问题，那么就可以使用广度优先搜索解决。

算法

`cover` 表示一条路径上访问过的节点集合，`head` 表示当前节点。在 `cover` 中使用比特位表示节点的访问情况，如果 `cover` 的第 k 个比特位是 1，表示该路径经过了第 k 个节点。

对当前 `state = (cover, head)` 使用广度优先搜索，从当前节点 `head` 出发到达每个邻接点 `child` 的新路径为 `(cover | (1 << child), child)`。

根据广度优先搜索可知，如果找到一个 `state` 包含了全部顶点，那么该 `state` 一定代表最短路径的长度。

Java | Python

```
class Solution(object):
    def shortestPathLength(self, graph):
        N = len(graph)
        queue = collections.deque((1 << x, x) for x in xrange(N))
        dist = collections.defaultdict(lambda: N*N)
        for x in xrange(N): dist[1 << x, x] = 0

        while queue:
            cover, head = queue.popleft()
            d = dist[cover, head]
            if cover == 2**N - 1: return d
            for child in graph[head]:
                cover2 = cover | (1 << child)
                if d + 1 < dist[cover2, child]:
                    dist[cover2, child] = d + 1
                    queue.append((cover2, child))
```

格雷编码

89. 格雷编码

难度 中等 200 收藏 分享 切换为英文 关注 反馈

格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。

给定一个代表编码总位数的非负整数 n ，打印其格雷编码序列。即使有多个不同答案，你也只需要返回其中一种。

格雷编码序列必须以 0 开头。

示例 1：

```
输入: 2
输出: [0, 1, 3, 2]
解释:
00 - 0
01 - 1
11 - 3
10 - 2
```

对于给定的 n ，其格雷编码序列并不唯一。
例如， $[0, 2, 3, 1]$ 也是一个有效的格雷编码序列。

```
00 - 0
10 - 2
11 - 3
01 - 1
```

示例 2：

```
输入: 0
输出: [0]
解释: 我们定义格雷编码序列必须以 0 开头。
给定编码总位数为  $n$  的格雷编码序列，其长度为  $2^n$ 。当  $n = 0$  时，长度为  $2^0 = 1$ 。
因此，当  $n = 0$  时，其格雷编码序列为 [0]。
```

特别想图，每一个数都连着 n 个与其二进制位只差 1 的数，但因为不能重复，所以需要剪枝。

```
class Solution:
    def grayCode(self, n: int) -> [int]:
        res, vis = [0], [0] # 题目中唯一要求从0开始，其余的顺序随便
        def dfs(cur): # 找到所有与cur二进制位唯一不同的数
            for i in range(n):
                nxt = cur ^ (1 << i) # 任何位与1异或得到相反数，与0异或得到本身
                if nxt in vis:
                    continue
                vis.append(nxt)
                res.append(nxt)
                dfs(nxt)
        dfs(0)
        return res
```

复杂数据结构

红黑树

定义

加入了限制条件的二叉排序树

红黑树四个特性要记牢：

- 1、首尾为黑（根节点、叶子节点为黑）
- 2、节点要么为黑、要么为红
- 3、不存在两个连续的红节点（红节点的子节点为黑）
- 4、从任一个节点到叶子节点出发的黑高相等

红黑树的插入

红黑树的插入

在一棵AVL树中，我们通过左旋和右旋来调整由于插入和删除所造成的不平衡问题。在红黑树中，可以使用两种方式进行平衡操作：

1. 重新着色
2. 旋转

当红黑树中出现不平衡的状态，我们首先会考虑重新着色，如果重新着色依旧不能使红黑树平衡，那么就考虑旋转。插入操作主要有两种情况，具体取决于叔叔结点的颜色。如果叔叔结点是红色的，我们会重新着色。如果叔叔结点是黑色的，我们会旋转或者重新着色，或者两者都考虑。

一个 NULL 结点被认为是黑色的，这在上篇已经提到过。

设 x 为新插入的结点。

1. 进行标准的 BST 插入并将新插入的结点设置为红色。
2. 如果 x 是根结点，将 x 的颜色转化为黑色（整棵树的黑高增加 1）。
3. 如果 x 的父结点 p 不是黑色并且 x 不是根结点，则：
 - a) 如果 x 的叔叔结点 u 是红色。
 - b) 如果 x 的叔叔结点 u 是黑色，则对于 x 、 x 的父结点 p 和 x 的爷爷结点 g 有以下四种情况：
 - LL (p 是 g 的左孩子且 x 是 p 的左孩子)
 - LR (p 是 g 的左孩子且 x 是 p 的右孩子)
 - RR (p 是 g 的右孩子且 x 是 p 的右孩子)
 - RL (p 是 g 的右孩子且 x 是 p 的左孩子)
 - 将插入结点 x 的父结点 (Parent) p 和叔叔 (uncle) 结点 u 的颜色变为黑色；
 - 将 x 的爷爷结点 (Grand Parent) g 设置为红色；
 - 将 x = x 的爷爷结点 g ，对于新的 x 重复 2, 3 两步。

总结：(因为插入的是红色节点，所以插入主要会导致连续的两个红色节点，排除它即可)

- 1、插入的节点都是红色节点，但插入的节点作为根节点，那么将其重新着色为黑色
- 2、若父节点是黑色，则不用重新着色和旋转

3、若父节点是红色：

A、叔叔节点是红色，那么只需将叔叔和父节点重新着色为黑色，将爷爷节点置为红色，那么视爷爷节点为新插入的节点，重复 2、3 步。

B、叔叔节点是黑色，结合爷爷节点和父节点的位置关系分为：

LL（右旋爷爷节点，交换新的爷爷节点和右孩子的颜色）

LR（先左旋父节点，转化为 LL）

RR（左旋爷爷节点，交换新的爷爷节点和左孩子的颜色）

RL（右旋父节点，转化为 LL）

综上：其实叔叔节点是黑色更好一点，因为不用重复 2、3 步。

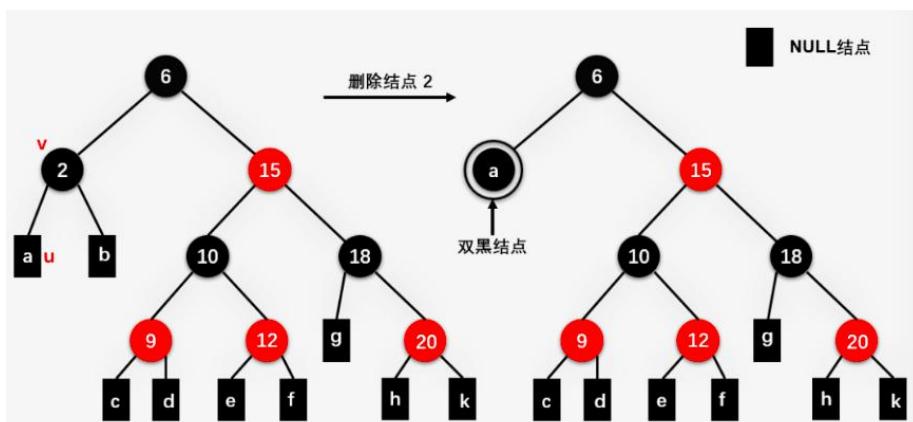
红黑树的删除操作

通过检查兄弟节点的颜色来决定恰当的平衡操作。

插入是为了避免红黑树中出现两个连续的红色节点，而删除操作是为了避免删除黑色节点导致黑高降低。

当删除的节点 v 是黑色节点，且其被黑色子节点替换时，其子节点就被标记为双黑：

删除操作的最主要任务就是将双黑节点转化为我们的普通黑色节点。



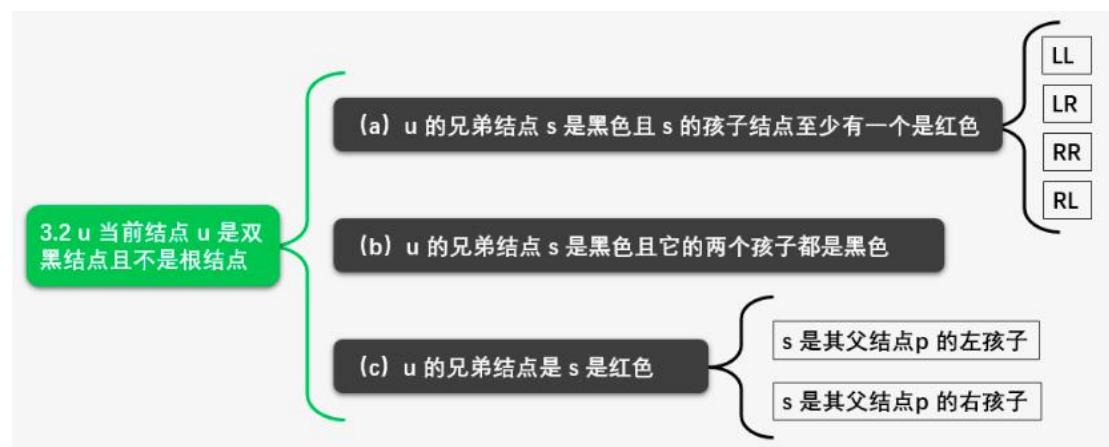
图中，为了方便，双黑节点直接变为了叶子节点，试想一下不是叶子节点的情况，你就

能想明白了。

既定删除节点为 v , u 是用来替换 v 的孩子节点：

删除操作总纲：

- 1、执行标准的 BST 的删除操作。
- 2、简单情况， u 或者 v 是红色
- 3、复杂情况： u 和 v 都是黑色节点



1、标准的 BST 删除操作：

值得一提的是，最终都会以删除一个叶子节点或者只有一个孩子的节点而结束。

2、简单情况： u 和 v 有一个是红色节点

将 u 替换 v ，并置黑

3、复杂情况：u 和 v 都是黑色节点

https://mp.weixin.qq.com/s?_biz=MzUyNjQxNjYyMg==&mid=2247489352&idx=5&sn=82d3ddf1c14cf9f5c81c76d2ae16c910&chksm=fa0e78c9cd79f1df1c2acf6506b40c8fe5a70c1f5e2f93fcc0c797db0cec596f91b46a1a04e1&mpshare=1&scene=1&srcid=&sharer.sharetime=1591142157907&sharer.shareid=af8c720bd8883efc17497f6364b732e0&exportkey=AwbTmPe07z0E7iuliebSWrU%3D&pass_ticket=QLz1%2FuywxgGgn3FQVbvaIyrwAu7Tmr5R7NISS32DB0YGsSTBIL0EtqASDWK1yd2R#rd

AVL 树

定义

加入了限制条件的二叉排序树

平衡二叉树

平衡二叉树又被称为 AVL 树，它是一棵二叉排序树，且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。

二叉排序树：是一棵空树，或者：若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；它的左、右子树也分别为二叉排序树。

树的高度：结点层次的最大值

平衡因子：左子树高度 - 右子树高度

红黑树VS平衡二叉树

	红黑树	平衡二叉树
相同点	都是二叉排序树	都是二叉排序树
查找效率	一般时间复杂度为 $O(\log N)$ ，最坏情况差于 AVL	时间复杂度稳定在 $O(\log N)$
插入效率	需要旋转操作和变色操作，插入结点最多只需要 2 次旋转；变色需要 $O(\log N)$ ；	插入结点最多只需要 1 次旋转， $O(\log N)$ 级别
删除效率	删除一个结点最多需要 3 次旋转操作	每一次删除操作最多需要 $O(\log N)$ 次旋转
优劣势	数据读取效率低于 AVL，维护性强于 AVL	数据读取效率高，维护性较差
应用场景	搜索，插入，删除操作差不多	搜索的次数远远大于插入和删除

红黑树和 AVL 树的区别：

红黑树的查询略逊于 AVL 树，但是红黑树的维护要简单一点，也就是多了着色的操作，那么每次插入和删除的平均旋转次数远小于 AVL 树。

1、红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。
2、平衡二叉树追求绝对平衡，条件苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知。

https://mp.weixin.qq.com/s?_biz=MzA4NDE4MzY2MA==&mid=2647521381&idx=1&sn=796ac1eda0eaefadfb57a1b9742bcec0&chksm=87d24766b0a5ce70a18acca20a130a14c16fb56a716d1c0elfbe0acf23915a1b8aad509f3850&mpshare=1&scene=1&srcid=&sharer_sharetime=1591660436390&sharer_shareid=af8c720bd8883efc17497f6364b732e0&exportkey=A9eUVXnxXFZbJ4D4,j4yaIGw%3D&pass_ticket=gT

[JqbSz%2Bbx6M62Kr1rg62d0%2FXQ%2Fkbd%2BCTCpk6cQDTB7mT9uQb%2FLcmogz9SQSR](#)

J3F#rd

[https://mp.weixin.qq.com/s?_biz=MzA4NDE4MzY2MA==&mid=2647521508&idx=1&sn=ff0751a1a49a48450757b53978fcbef8&chksm=87d247e7b0a5cef1f5f581cfa843b68021a51e979ee49b2b947cf394c613b4701ac07a8e8a76&mpshare=1&scene=1&srcid=&sharer.sharetime=1591660449695&sharer.shareid=af8c720bd8883efc17497f6364b732e0&exportkey=A3d7cLTEicDt1tQY1TCDPNI%3D&pass_ticket=gT">JqbSz%2Bbx6M62Kr1rg62d0%2FXQ%2Fkbd%2BCTCpk6cQDTB7mT9uQb%2FLcmogz9SQSR](https://mp.weixin.qq.com/s?_biz=MzA4NDE4MzY2MA==&mid=2647521508&idx=1&sn=ff0751a1a49a48450757b53978fcbef8&chksm=87d247e7b0a5cef1f5f581cfa843b68021a51e979ee49b2b947cf394c613b4701ac07a8e8a76&mpshare=1&scene=1&srcid=&sharer.sharetime=1591660449695&sharer.shareid=af8c720bd8883efc17497f6364b732e0&exportkey=A3d7cLTEicDt1tQY1TCDPNI%3D&pass_ticket=gT)

J3F#rd

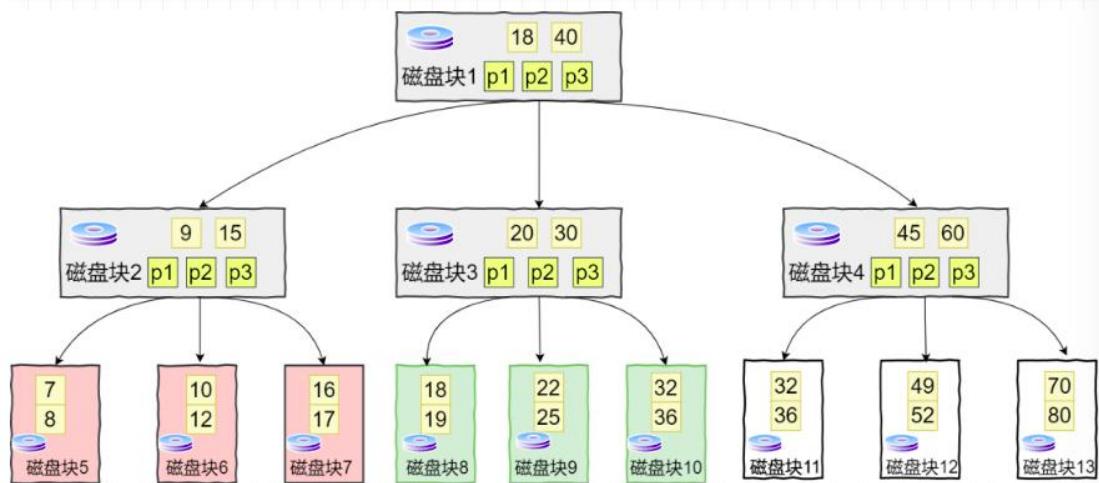
B 树

B 树的主要目的是减少磁盘的 IO 操作，一般远小于 logn。那既然这样，为啥还要红黑树和 AVL 树呢，原因我认为是维护不易，页分裂和页合并是二叉树不具备的劣势。

M 阶 B 树

- 1、除根节点和叶子节点外，每个节点有 t 个关键字， $t+1$ 个孩子节点 ($\text{ceil}(M/2) / 2 < t < M$)。根节点的儿子数量范围为 $[2, M]$ 。
- 2、每个节点都会存数据，关键字是 key，也就是说 val 也会存在节点上。
- 3、关键字升序排列
- 4、叶子节点位于同一层

B树的容颜



https://mp.weixin.qq.com/s?_biz=MzA4NDE4MzY2MA==&mid=2647522005&idx=1&sn=659962d777276bbd16a581ddc884c69d&chksm=87d245d6b0a5ccc0f75159f1c4bcc3462f760e802ced693ac78808dfcbfe0d4b815da3f9b584&mpshare=1&scene=1&srcid=&sharer_sharetime=1591421935992&sharer_shareid=af8c720bd8883efc17497f6364b732e0&key=fc29ef84290ceb7414f5cf789fbff33ccb41d551611f7eb468547229dd830e6e4a4739faece5a358daeb8aefcaf5d1799eaab727522ee86019d39f7f40bccd4ced58a123789323a52285b20087765564&ascene=1&uin=MTU0MDc2NTcy0Q%3D%3D&devicetype=Windows+10+x64&version=62090070&lang=zh_CN&exportkey=A9oDmiwsz%2BA0i7sw%2FX3EOFw%3D&pass_ticket=uni5FA3MKTsbX5N87rUez8oD1%2BieIQH01y9jeT04NH6RgXgm1BzZ1CWJQIrmiUb0

B+树

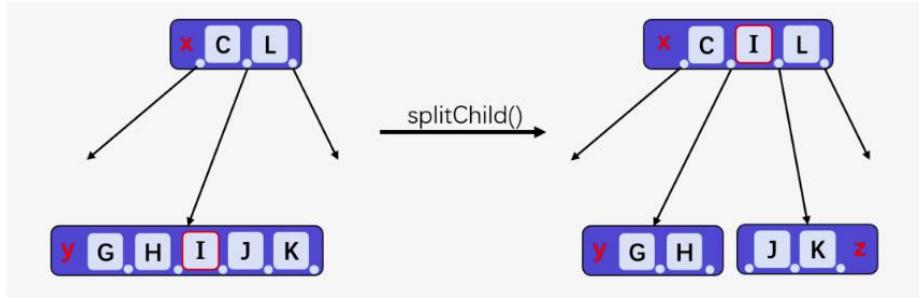
- 1、孩子数量 == 关键字数量 (原因是非叶子节点不存数据，所以非叶子节点只存key，不存val)
- 2、所有关键字都存在叶子节点，构成双向有序链表。

页分裂操作

当插入一个值后，超过了 mysql 页的大小，则会将中间节点上移到父节点，也称为成
长，然后分裂。

如果判断在插入一个关键字 k 之前，一个结点是否可供当前结点插入的空间呢？

我们可以使用一个称为 `splitChild()` 的操作实现，即拆分一个结点的孩子。下图中，`x` 的孩子结点 `y` 被拆分成了两个结点 `y` 和 `z`。拆分操作将一个关键 I 上移，并以上移的关键 I 对结点 `y` 进行拆分，拆分成包含关键字 $[G, H]$ 的结点 `y` 和包含关键字 $[J, K]$ 的结点 `z`。这一过程又称之为 B-树的生长，区别于 BST 的向下生长。



综上，B-树在插入一个新的关键字 k 时，我们从根结点一直访问到叶子结点，在遍历一个结点之前，首先检查这个结点是否已经满了，即包含了 $2t - 1$ 个关键字，如果结点已满，则将其拆分并创建新的空间。插入操作的伪代码描述如下：

B 树和 B+ 树对磁盘极度友好，而对内存可能友好的不明显：

树矮胖，避免寻道次数，降低 IO

B+ 树相比 B 树的优势：

查询效率更稳定，都是 Ologn

遍历所有叶子节点即可实现整棵树便利，而 B 树，需要中序遍历达到相同效果。

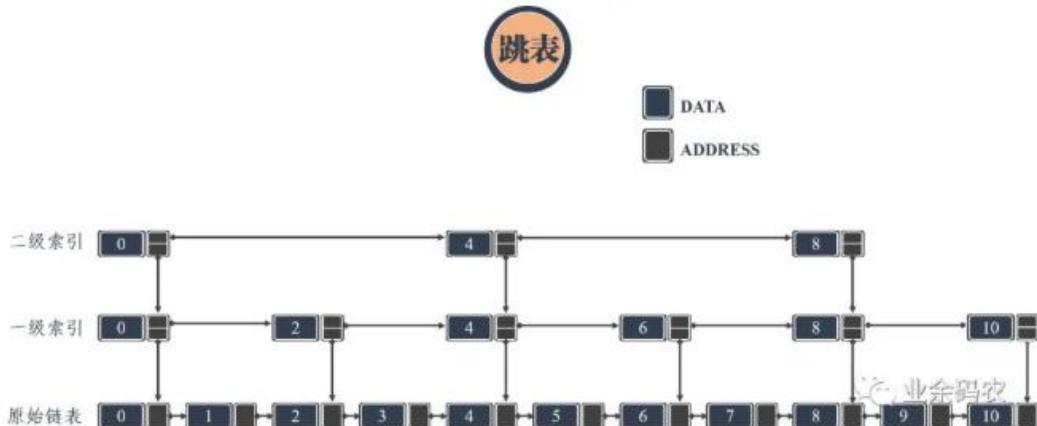
更矮胖，查询更快。

https://mp.weixin.qq.com/s?_biz=MzA4NDE4MzY2MA==&mid=2647522161&idx=2&sn=b2d5e043159569928e77dd2e2836ab07&chksm=87d24472b0a5cd6407f2092958029505e2b51c52f7cdb7318dd5a98bbd9dfaf89c87bed87dfa&mpshare=1&scene=1&srcid=&sharer.sharetime=1592649909697&sharer.shareid=af8c720bd8883efc17497f6364b732e0&exportkey=A9E6tpNJ.jCNMkLXN8,j7rpKo%3D&pass_ticket=gLMz1It2yz2UHk6Ud0mTqQk0%2Fctkmue2Fs%2BD1VPaKLrbwUwfQg1RHVdDCv3vMED#rd

跳表

3 跳表

从上面的对比中可以看出，链表虽然通过增加指针域提升了自由度，但是却导致数据的查询效率恶化。特别是当链表长度很长的时候，对数据的查询还得从头依次查询，这样的效率会更低。**跳表的产生就是为了解决链表过长的问题**，通过增加链表的多级索引来加快原始链表的查询效率。这样的方式可以让查询的时间复杂度从 $O(n)$ 提升至 $O(\log n)$ 。



跳表通过增加的多级索引能够实现高效的动态插入和删除，其效率和红黑树和平衡二叉树不相上下。目前 redis 和 levelDB 都有用到跳表。

从上图可以看出，索引级的指针域除了指向下一个索引位置的指针，还有一个 **down** 指针指向低一级的链表位置，这样才能实现跳跃查询的目的。

处理输入输出

Input()处理单行，多行数据

```
class Solution():
    def print(self, list):
        for x in list:
            print(x)

if __name__ == '__main__':
    solution = Solution()
    n1, n2 = map(int, input("请输入两个数:").strip().split()) # 单行接收已知个参数

    # line = list(map(int, input("请输入一行:").strip().split())) # 单行将一行数字存入列表

    # line = input("请输入一行:").strip().split() # 单行将一行字符串存入列表

    mx = [] # 建立二维列表
    for _ in range(n2): # 确认n2是行数
        string = input().strip()
        lineList = list(map(int, string.split()))
        mx.append(lineList)
    print(mx)
```

进制转换

写出一个程序，接受一个十六进制的数，输出该数值的十进制表示。（多组同时输入）

```
while True:
    try:
        print(int(input(), 16)) # 后面的16表示将input输入的字符识别成16进制的数，然后进行化整
    except:
        break
```

字符个数统计

题目描述

编写一个函数，计算字符串中含有的不同字符的个数。字符在ASCII码范围内(0~127)，换行表示结束符，不算在字符串里。不在范围内的不作统计。注意是不同的字符。

输入描述:

输入N个字符，字符在ASCII码范围内。

输出描述:

输出范围在(0~127)字符的个数。

示例1

输入	复制
abc	
输出	复制
3	

思路： `ord()`函数主要用来返回对应字符的 ascii 码：也就是字符变成数字

```
print ord(a)
#97
```

`chr()`主要用来表示 ascii 码对应的字符：也就是数字变成字符，他的输入是数字，

可以用十进制，也可以用十六进制。

```
print chr(97)
#a
print chr(0x61)
#a

if __name__ == '__main__':
    s = input()
    result = [x for x in s]
    result = set(result)
    count = 0
    for x in result:
        if 0 <= ord(x) < 128: # ord函数用来将ascii
            count += 1
    print(count)
```

面试题汇总

智力题

1、动物划船

大老虎,小老虎,大豹子,小豹子,大狼狗,小狼狗:三对母子要过河,河边只有一条船,船每次只能装两个(不论大小),其中只有三个大的和小老虎四个会划船,但每一种动物小的不能单独呆在一边否则会被别一种类大的吃掉,请问你怎样划船才能让它们都安全过河?

诀窍就是： 将 ABC 先过河，然后 a 返回去接 b、c

1、Aa 过河

2、BC 过河

3、a 返回

4、ab 过河

5、ac 过河

老鼠试毒

1、高频面试题：老鼠试毒

有 8 个一模一样的瓶子，其中有 7 瓶是普通的水，有一瓶是毒药。任何喝下毒药的生物都会在一星期之后死亡。现在，你只有 3 只小白鼠和一星期的时间，如何检验出哪个瓶子里有毒药？

解题步骤如下：

1、把这 8 个瓶子从 0 到 7 进行编号，用二进制表示如下

```
000  
001  
010  
011  
100  
101  
110  
111
```

2、将 0 到 7 编号中第一位为 1 的所有瓶子（即 1, 3, 5, 7）的水混在一起给老鼠 1 吃，第二位值为 1 的所有瓶子（即 2, 3, 6, 7）的水混在一起给老鼠 2 吃，第三位值为 1 的所有瓶子（4, 5, 6, 7）的水混在一起给老鼠 3 吃，现在假设老鼠 1, 3 死了，那么有毒的瓶子编号中第 1, 3 位肯定为 1，老鼠 2 没死，则有毒的瓶子编号中第 2 位肯定为 0，得到值 101，对应的编号是 5，也就是第五瓶的水有毒。

这道题及其相关的变种在面试中出现地比较频繁，比如我现在把 8 瓶水换成 1000 瓶，问你至少需要几只老鼠才能测出有毒的瓶子，有了上述的思路相信应该不难，几只老鼠就相当于几个进制位，显然 $2^{10} = 1024 > 1000$ ，即 10 只老鼠即可测出来。

小白鼠的数量少于毒药瓶数，所以小白鼠的数量等于二进制位的数量，对应二进制为 1 的毒药，都给该小白鼠喝，小白鼠死亡，表示毒药的该二进制位一定含有 1.

华为：

待：景点间最短路径实时查询系统

[编程|300分] 景点间最短路径实时查询系统

时间限制：C/C++ 2秒，其他语言 4秒

空间限制：C/C++ 262144K，其他语言 524288K

64bit IO Format: %lld

本题可使用本地IDE编码，不做跳出限制，编码后请点击“保存并调试”按钮进行代码提交。

■ 题目描述

某旅游景区有n个景点，编号从1到n，其中景点1还是游客集散中心，景点之间相隔较远，有 $2n-2$ 条单向车道连接，编号从1到 $2n-2$ ，车道可以分为两大类：

1、前 $n-1$ 条单向车道会组成一个以景点1（集散中心）为根的生成树（根节点直达任何节点，部分直达，部分通过其他节点跳转可达），方便从集散中心到达各个景点。

2、后 $n-1$ 条单向车道从景点*i*指向景点1（集散中心），其中 $2 \leq i \leq n$ ，也就是其他节点都可以直达根节点，方便用户随时从各景点返回。

现在需开发一个系统用于实时查询两个景点之间的最短距离，系统支持两种操作命令：

1、操作命令为1 *i w*，1表示修改操作，表示将第*i*条车道的长度调整为*w*

2、操作命令为2 *u v*，2表示查询操作，表示打印景点*u*到景点*v*的最短路径

根据所有的操作命令，打印所有查询操作的结果。

输入描述：

输入包括三部分：

1、第一行包含两个整数*n*和*q*，*n*表示景点的个数，*q*表示操作命令的个数。

2、中间 $2n-2$ 行表示所有的单向车道，每一行包含三个整数 a_i, b_i, c_i ，表示景点 a_i 到景点 b_i 的距离为 c_i ，

其中前 $n-1$ 条单向车道会组成一个以为景点1（集散中心）为根的生成树，后面 $n-1$ 条单向车道 $b_i=1$ ，表示景点*i*到景点1（集散中心的）距离， $2 \leq i \leq n$ 。

3、最后*q*行表示*q*个操作命令。

条件限制：

$2 \leq n \leq 1000$ ；

$1 \leq q \leq 1000$ ；

$1 \leq a_i, b_i \leq n$ ；

单向车道的长度*w*: $1 \leq w \leq 1000$

输出描述：

根据所有的操作命令，打印所有查询操作的结果。

输入：

6 6
1 3 4
3 2 5
1 4 6
4 5 2
4 6 3
2 1 3
3 1 4
4 1 3
5 1 5
6 1 2
2 1 5
2 6 5
1 4 3
2 1 6
2 6 5
2 3 5

输出：

8
10
9
11
13

```

class Solution():
    def minLen(self, mx, n, g):
        NameByNo = dict()
        chedaoByName = dict()
        N = len(mx)
        for i in range(2 * n - 2):
            key = mx[i][0] + mx[i][1]
            val = int(mx[i][2])
            NameByNo[i + 1] = key
            chedaoByName[key] = val
        print(chedaoByName)
        for j in range(2 * n - 2, N):
            if int(mx[j][0]) == 1:
                key = NameByNo[int(mx[j][1])]
                chedaoByName[key] = mx[j][2]
            else:
                key = mx[j][1] + mx[j][2]
                if key in chedaoByName: # 此处应该用图的遍历算法，可惜没时间了
                    print(chedaoByName.get(key))
                else:
                    print(0)

if __name__ == '__main__':
    solution = Solution()
    n1, n2 = map(int, input().strip().split())
    N = 2 * n1 - 2 + n2
    mx = []
    for _ in range(N):
        string = input().strip()
        lineList = string.split()
        mx.append(lineList)
    solution.minLen(mx, n1, n2)

```

Case 0%

待：视频会议室使用时长最大化

[编程|200分] 视频会议室使用时长最大化

时间限制：C/C++ 1秒，其他语言 2秒

空间限制：C/C++ 32768K, 其他语言 65536K

64bit IO Format: %lld

本题可使用本地IDE编码，不做跳出限制，编码后请点击“保存并调试”按钮进行代码提交。

■ 题目描述

公司2楼有一个视频会议室，会议室的使用由秘书进行安排，每个使用会议的小组都会发送会议使用时间 (start, end) 给秘书，时间用24小时制表示，会议最早开始时间8点，最晚23点结束；会议都是整数小时，上一场会议的结束时间和下一场会议的开始时间可以相同，不考虑会议的重要紧急程度，会考虑会议延迟，作为秘书安排会议的原则就是让会议室的使用时间最长；

输入描述：

输入一个数字T，表示有T组测试数据；每个测试数据输入一个数字n ($1 < n \leq 100$)，表示几场会议；然后紧跟着n行数字对，每行分别表示会议开始和会议结束

输出描述：

输出会议室最长使用时间，每组样例输出最终结果，并且单独占用一行

输入：

```
1
6
15 17
8 11
10 16
11 12
13 15
9 12
```

输出：

```
8
```

```

class Solution():
    def print(self, Tmx):
        for mx in Tmx:
            N = len(mx)
            mx = sorted(mx, key=lambda x: (x[0], -x[1]))
            self.Time = 0
            self.dfs(mx, 0, 8)
            print(self.Time)

    def dfs(self, mx, time, startTime): # mx的最长使用时长
        N = len(mx)
        if N == 0:
            self.Time = max(self.Time, time)
            return
        for i in range(N):
            if mx[i][0] >= startTime:
                time += mx[i][1] - mx[i][0]
                self.dfs(mx[i + 1:], time, mx[i][1])
                time -= mx[i][1] - mx[i][0]

    if __name__ == '__main__':
        solution = Solution()
        N = int(input())
        Tmx = []
        for _ in range(N):
            lineNum = int(input()) # 单行将一行数字存入列表
            mx = [] # 声明二维列表
            for _ in range(lineNum):
                string = input().strip()
                lineList = list(map(int, string.split()))
                mx.append(lineList)
            Tmx.append(mx)
        solution.print(Tmx)

```

Case 10%

待：跳跳棋

[编程|100分] 跳跳棋

时间限制: C/C++ 1秒, 其他语言 2秒

空间限制: C/C++ 32768K, 其他语言 65536K

64bit IO Format: %lld

本题可使用本地IDE编码, 不做跳出限制, 编码后请点击“保存并调试”按钮进行代码提交。

■ 题目描述

现在有一种跳跳棋, 跳棋路线有N格排列为一条直线, 起始位置和结束位置都在棋盘之外, 跳到每一格上都可以获取一定的积分, 但是不可以从一格跳到相邻的格, 也不可以回跳, 请问如何获取最高的积分

输入描述:

第一行为整数N, 表示跳棋格数 ($1 \leq N \leq 100000$)

第二行为每一格代表的分数M ($1 \leq M \leq 1000$)

输出描述:

能获得的最高积分

1、子问题规划 : $dp[n]$ 表示跳到 $nums[i]$ 的最高积分

2、Base case : $dp[0] = nums[0]$

3、状态转移 :

$$Dp[i] = \max(dp[i-1], dp[i-1]+nums[i])$$

示例1 输入输出示例仅供调试，后台判题数据一般不包含示例

输入

```
3  
1 5 2
```

复制

输出

```
5
```

复制

示例2 输入输出示例仅供调试，后台判题数据一般不包含示例

输入

```
4  
5 2 4 9
```

复制

输出

```
14
```

复制

```
class Solution():
    def jump(self, line):
        N = len(line)
        # dp = [0] * N # dp[i] 表示跳到第i格获取的最高积分
        # dp[0] = 0
        pre, cur = 0, 0
        for i in range(N):
            # dp[i] = max(dp[i - 1], dp[i - 2] + line[i])
            pre, cur = cur, max(cur, pre + line[i])
        # print(line)
        # print(dp)
        print(cur)

if __name__ == '__main__':
    solution = Solution()
    N = int(input())
    line = list(map(int, input().strip().split())) # 单行将一行数字存入列表
    solution.jump(line)
```

Case 60%

字节

- 1、求数组中两数的最小差
- 2、检查是否是有效括号，大括号在外面，中括号在中间，小括号在里面
- 3、一个数，去除 n 位，保证该数最大